conference

proceedings

13th Systems Administration Conference (LISA '99)

Seattle, Washington, USA
November 7-12, 1999

Co-Sponsored by

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

SAGE
THE SYSTEM ADMINISTRATORS GUILD

USENIX Proceedings of the 13th Systems Administration Conference (LISA '99)

Seattle, Washington, USA, November 1999

For additional copies of these proceedings contact

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Telephone: 510-528-8649
http://www.usenix.org
<office@usenix.org>

The price is $32 for members and $40 for nonmembers.

Past USENIX Large Installation Systems Administration Workshop and Conference Proceedings (price: member/nonmember)

Large Installation Systems Admin. I Workshop	1987	Phildelphia, PA	$4/$4
Large Installation Systems Admin. II Workshop	1988	Monterey, CA	$8/$8
Large Installation Systems Admin. III Workshop	1989	Austin, TX	$13/$13
Large Installation Systems Admin. IV Conference	1990	Colorado Spgs, CO	$15/$18
Large Installation Systems Admin. V Conference	1991	San Diego, CA	$20/$23
Systems Administration VI Conference	1992	Long Beach, CA	$23/$30
Systems Administration VII Conference	1993	Monterey, CA	$25/$33
Systems Administration VIII Conference	1994	San Diego, CA	$22/$29
Systems Administration IX Conference	1995	Monterey, CA	$30/$38
Systems Administration X Conference	1996	Chicago, IL	$30/$38
Systems Administration XI Conference	1997	San Diego, CA	$30/$38
Systems Administration XII Conference	1998	Boston, MA	$32/$40

Outside the U.S.A. and Canada, please add $12 per copy for postage (via air printed matter).

ISBN 1-880446-25-1

♻ Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

Proceedings of the Thirteenth Systems Administration Conference (LISA XIII)

November 7-12, 1999
Seattle, WA, USA

TABLE OF CONTENTS

Thinking on the Job

Thursday (9:00 - 10:30 am) — Chair: Thomas A. Limoncelli

Network Infrastructure

Thursday (11:00 am - 12:30 pm) — Chair: Greg Rose

File Systems

Thursday (2:00 - 3:30 pm) — Chair: Doug Kingston

Systems

Thursday (4:00 - 5:30 pm) — Chair: Eric Anderson

Network Security

Friday (9:00 - 10:30 am) — Chair: Jeffrey A. Uphoff

Installations

Friday (2:00 - 3:30 pm) — Chair: Evi Nemeth

ACKNOWLEDGMENTS

PROGRAM CHAIR
David Parter, *University of Wisconsin*

PROGRAM COMMITTEE
Eric Anderson, *University of California, Berkeley*
Strata Rose Chalup, *VirtualNet Consulting*
Xev Gittler, *Goldman, Sachs & Co.*
Trent Hein, *XOR Network Engineering, Inc.*
Doug Kingston, *Deutsche Bank*
Thomas A. Limoncelli, *Lucent Technologies/Bell Labs*
Bryan McDonald, *Global Networking And Computing*
Adam Moskowitz, *LION Biosciences Research, Inc.*
Evi Nemeth, *University of Colorado*
Cat Okita, *Earthworks*
Josh Simon, *Collective Technologies*
Jeffrey A. Uphoff, *Transmeta Corporation*

READERS
William S. Annis, *University of Wisconsin*
Gus Hartmann, *AT&T Wireless Services*
Doug Hughes, *Auburn University*
Georg Nikodym, *Sun Microsystems*
Kathryn L. Penn, *University of Maryland*
Erez Zadok, *Columbia University*

INVITED TALKS COORDINATOR
Phil Scarr, *GE-FANUC*

PRACTICUM CHAIR
Pat Wilson, *Dartmouth College*

CONFERENCE ADVISOR
Rob Kolstad, *The SANS Institute*

UNIVERSITY SYSADMIN EDUCATION
John Sechrest, *PEAK, Inc.*
David Kuncicky, *Bioreason, Inc.*

GIGA LISA
Joel Avery, *Nortel Networks*

ADVANCED TOPICS
Adam Moskowitz, *LION Biosciences Research, Inc.*

GURU-IS-IN COORDINATOR
Lee Damon, *QUALCOMM, Inc.*

TERMINAL ROOM COORDINATOR
Lynda McGinley, *University of Colorado*

PROCEEDINGS PRODUCTION
Rob Kolstad, *The SANS Institute*
Data Reproductions

USENIX ASSOCIATION
Ellie Young, *Executive Director*
Judith F. DesHarnais, *Conference Director*
Daniel V. Klein, *Tutorial Coordinator*
Gale Berkowitz, *Deputy Executive Director*
Monica Ortiz, *Marketing Communications Manager*
Jane-Ellen Long, *Publications Director*

USENIX SUPPORT STAFF
Bleu Castañeda, *USENIX Association*
Moun Chau, *USENIX Association*
Cami Edwards, *USENIX Association*
Vanessa Fonseca, *USENIX Association*
Dana Geffner, *USENIX Association*
Jennifer Radtke, *USENIX Association*
Toni Veglia, *USENIX Association*

PREFACE

Welcome to LISA '99!

We've assembled a program that includes 27 of the best papers, the traditional Invited Talks track, and a return of last year's Practicum track. The program features outstanding speakers, timely topics, and a bit of fun for everyone.

The Conference Proceedings contain the printed papers of the refereed track. The proceedings of the LISA conferences represent the historical record of the development of our field in both technology and methodology. It is through these papers and the other activities at the LISA conferences that the field as a whole shares ideas and knowledge, and therefore advances. The broad range of topics included in the program show the wide variety of concerns and challenges systems administrators face today and are anticipating in the future.

The Program Committee had a daunting task in selecting the papers. Thanks to all who contributed: those who took time from their always-crowded schedule to write and submit abstracts; the program committee and outside reviewers; and of course the authors of the accepted papers who devoted even more time to writing the full papers and preparing the presentations.

This year's Invited Talk track was assembled by Phil Scarr, and the Practicum was coordinated by Pat Wilson. Both Phil and Pat have put together great tracks that complement each other and the refereed papers to provide a diverse and interesting technical program.

There are many other activities at LISA that help to make the conference what it is: the "Guru Is In", the Birds-of-a-Feather Sessions, and the terminal room, to name a few. All of these, and many others, are made possible by the many volunteers who deserve our thanks.

Every year, the Program Chair thanks the USENIX staff for their great work on the conference. Until now, I did not know how true those accolades were – the conference would simply not be possible without the hard work and dedication of Judy DesHarnais, the USENIX Conference Director; Ellie Young, the USENIX Executive Director; and the staff of the Conference and Association offices. In addition, Rob Kolstad and Greg Rose provided a great deal of helpful and timely advice. I would be remiss if I did not thank my co-workers and friends for the advice and help they provided. Thank you all!

Thanks for coming to Seattle! See you in the hallways!

David Parter

AUTHOR INDEX

ssmail: Opportunistic Encryption in *sendmail*

Damian Bentley – Australian National University†
Greg Rose – QUALCOMM Australia
Tara Whalen – Communications Research Centre Canada

ABSTRACT

Much electronic mail is sent unencrypted, making it vulnerable to passive eavesdropping. We propose to protect email privacy by building encryption functionality into ESMTP mailers. Our solution, ssmail, provides fast, simple encryption for sendmail that does not require user intervention or reliance on public key infrastructure. We added a small number of steps to an ESMTP session, thereby allowing a client and server to create a secret, one-time session key used to encrypt the mail transfer session. ssmail relies on caching to reduce key generation overhead. The overhead imposed by our encryption scheme is minimal, allowing even busy mail servers to support privacy.

We placed our encryption mechanism within the mail transfer agent itself, allowing people to use privacy protection software without having to know how to run an encryption program explicitly. Furthermore, we are able to encrypt the email transmission session, protecting such information as sender and recipient identities. The speed and simplicity of ssmail make it a very useful addition to widely deployed ESMTP mailers. Our solution can also be adopted easily by other mailers, and can be extended to use other encryption algorithms.

Introduction

A great deal of email travelling over the Internet is vulnerable to eavesdropping. Eavesdropping operations can be done in bulk by well-funded adversaries such as organized crime or government agencies. While email is in transit from sender to receiver, it usually travels over a number of paths, any of which might be tapped by a passive listener. Encrypting email is the obvious solution to this threat. A properly used encryption algorithm is the most effective way to ensure privacy against the casual interceptor.

However, there are a number of factors that can make encryption an impractical solution. Some users are unaware of encryption technology, or how to use it properly. Furthermore, encryption algorithms can be computationally intensive, rendering them unfit for overburdened mail servers. Lastly, end to end encryption of email requires a pervasive key management infrastructure and, despite some progress, no such infrastructure exists today.

For these reasons, almost all email is currently sent unencrypted over the Internet. Our goal for this work is to significantly increase the proportion of electronic mail that is encrypted and protected from bulk eavesdropping. It is *not* our goal to make email transmission totally secure – this is better done at the application layer using, for example, PGP [19].

To address these issues, we implemented ssmail, a patch for sendmail[1] that allows for the encryption of mail sent using the Extended Simple Mail Transfer Protocol (ESMTP) [8]. By adding a few extra steps to an ESMTP session, we allow a client and server to exchange enough information to create a secret, one-time session key that is used to encrypt the mail transfer session. Our solution includes careful use of caching to reduce key generation overhead. By placing encryption within the mail transfer agent itself, we allow users to take advantage of privacy protection without having to run an encryption program explicitly. An added benefit is that all of the transmission session, including such things as sender and recipient identities, is encrypted.

The speed and simplicity of this solution make it a very useful addition to widely deployed ESMTP mailers.

Design goals

Our central goal when developing ssmail was to provide a fast, easy to use method to protect email from eavesdroppers. Note that a single network tap at a hub location could yield an enormous amount of information. Owing to the great deal of email that is transmitted and processed every day, any encryption method used must be fast. A complex encryption scheme, or one that requires a trusted third-party machine to distribute keys, will slow communication. In order to make encryption easy for users, we did not want to rely on public-key infrastructure or require that users know how to use encryption software properly. Rather, we wanted the encryption to be effectively transparent to users. They can determine that it

†The author worked on this project while employed with QUALCOMM Australia.

was used if they examine the mail headers; they will not, however, have to make it work themselves.

The Threat Model

Before describing ssmail in detail, it is necessary to clarify the threat model against which it defends. Email messages typically make their way from the sender to the recipient in a number of hops. The first hop is often inside an organization, such as a large company or an Internet service provider, and might be presumed to be secure. Similarly, the last hop might be made from an organization's firewall to the receiving machine. However, the hop(s) in the middle are usually across the Internet: they are insecure and thus perfect targets for mass interception.

Current implementations of ESMTP transmit all commands and email messages in the clear, making email vulnerable as it crosses many machines across a network. Our goal was to provide efficient protection of email privacy in the face of *passive* attacks. Our solution defends against passive attacks: anyone intercepting an ESMTP session that uses ssmail will be unable to discover the sender, receiver, or email contents.

However, ssmail *cannot* (and does not try to) defend against active attacks. The Diffie-Hellman key exchange used in ssmail does not authenticate the participants. An interceptor could substitute her public key for the real recipient's key, allowing her to read and possibly modify email before re-encrypting it with the real key and sending the email on its way. The sender and receiver will have no idea that their supposedly safe transmission was intercepted.

Defeating such an active attack is not an easy task. Solutions involve the use of a public key infrastructure (PKI), preferably one that uses certificates that ensure that public keys cannot be tampered with. The Station-to-Station protocol [10], for example, uses a PKI to allow the participants to verify that they received the correct keys during the Diffie-Hellman exchange. As mentioned above, there does not yet exist a pervasive key management infrastructure. When a PKI is in place, ssmail could be modified to perform the Station-to-Station protocol if authentication were deemed necessary. (Of course, when such a PKI is in place, interim solutions such as ssmail should not be necessary at all.) In this protocol, it is assumed that each party has access to the other party's public key; the "man in the middle" attack is foiled through the use of digital signatures. Such a protocol could offer stronger security with minimal added overhead, depending on how the PKI operates and how expensive it would be to obtain public keys securely or to send public-key certificates.

What ssmail *does* provide right now is a solution that is not risk-free, but is fast and simple, and can be deployed immediately to thwart the problem of passive listening.

Possible Solutions

There are a number of encryption methods available to email users that could guard against eavesdropping. People can use secondary programs such as PGP or S/MIME [4] to encrypt and decrypt their mail messages. The advantage of this approach is that users can select the encryption application that best serves their needs (such as speed or level of security). However, it requires that users be capable of using these applications correctly, which may be a problem for inexperienced users. A recent study indicates that this is more problematic than might be expected [17]. Furthermore, such methods do nothing to disguise the source or destination of the email, or any of the information in mail headers.

Another solution would be to rely on infrastructure that encrypts all traffic (including email) travelling over the network. IPSec and IPv6 [2] provide such a solution by supporting encryption of all IP packets. The advantages are obvious: all IP traffic is protected without the user having to use an encryption package directly. The major disadvantage is that such infrastructure is not yet widely deployed.

Our solution was to add an encryption command to ESMTP. This command provides mail agents with shared secret keys through a Diffie-Hellman exchange; these keys can then be used in encrypting email. We added an encryption scheme to sendmail that makes use of these shared keys. As described below, speedy key agreement is supported through the use of a special cache. ssmail users do not have to make any special effort to ensure that encryption works. Thus we provide fast and simple encryption support for ESMTP mailers.

Our Solution: *ssmail*

Communication between an ESMTP client and server consists of a series of commands and responses that are sent along with the email message. We added a new command to this protocol: the XCRYPT command. This command has a number of parameters that allow a client and server to create secret keys.

The input and output functions of sendmail are modified to encrypt and decrypt protocol messages sent. Below, we describe in detail how the XCRYPT command is used, how the keys are created, and how efficiency is addressed using caching.

The XCRYPT Command

We added the XCRYPT command to ESMTP to allow for the exchange of keys for email encryption. It has a number of parameters:

- **version strings**: the client and server exchange strings, which indicate the version and name of the preferred encryption algorithm(s) to be used (for example, t32_1.0)
- **public keys**: the client and server exchange public keys to be used in the Diffie-Hellman key agreement scheme

- **nonces**: both the client and server generate random integers that are hashed with the shared secret key to produce a new session key for each exchange (details below).

An ESMTP session that supports XCRYPT is shown in Table 1. As the table shows, the XCRYPT exchange requires five extra messages to be added to a normal ESMTP session. When a connection is made between two machines that have the ssmail patch installed, the exchange takes place as follows:

1. The server sends the initial introduction, which contains such information as the machine name and the version of sendmail being used.
2. The client sends an EHLO (extended hello) command along with its address. Sending EHLO (rather than SMTP's HELO) indicates that the client sendmail uses ESMTP.
3. The server responds to EHLO by providing a list of the ESMTP services that it supports, using XCRYPT to indicate encryption. It also sends the version string (V_b), which reports its available encryption algorithms.
4. The client then provides its version string (V_a), which includes the chosen encryption algorithm, along with its public key (K_a) and nonce (N_a). This provides the server with enough information to create the shared secret key used for encryption (detailed below).
5. The server sends its public key (K_b) and nonce (N_b), which the client uses to create the shared secret key.
6. If there are no problems with the exchange, the client sends XCRYPT OK. Failure is indicated by XCRYPT FAIL. Should a FAIL be sent in either direction, the session will continue, but the mail will be sent unencrypted.
7. If there are no problems on the server side, the server sends XCRYPT OK (else XCRYPT FAIL).

If the server is not capable of using the XCRYPT command, it is not mentioned in the list of supported services. Thus, the client will not respond with its XCRYPT command. Should the client be unable to support the XCRYPT command, this service (as sent by the server) is ignored, and the session continues normally, without encryption.

Creating Shared Secret Keys

Once the necessary information has been exchanged via the XCRYPT commands in ESMTP, the client and server use it to calculate shared secret keys used in encrypting the mail transfer. There are a number of components used in the calculation of the encryption keys, shown in Table 2.

The shared secret is generated using the Diffie-Hellman key agreement [3]. The client a and the server b share a strong prime p and a generator g. The prime and generator are hard-coded into ssmail – they

Client(a)		Server(b)	Notes
	←	220 (address)	1. Server introduction
EHLO (address)	→		2. Client introduction
	←	250 XCRYPT V_b	3. Server sends services it supports. Includes XCRYPT with version string.
XCRYPT V_a K_a N_a	→		4. Client sends version string, public key, nonce
	←	250 XCRYPT K_b N_b	5. Server sends public key, nonce
XCRYPT OK	→		6. Client XCRYPT verification
	←	250 XCRYPT OK	7. Server XCRYPT verification
Encryption starts now			

Table 1: An ESMTP session using XCRYPT.

Variable/Key	Public/Private	Notes
strong prime (p)	public	768-bit integer (static, shared)
generator (g)	public	value = 2 (static, shared)
private key (r)	private	randomly-chosen 768-bit integer
public key (K)	public	$K = g^r \pmod p$
shared secret	private	secret = $K^r \pmod p$
nonces (n)	public	randomly-chosen 80-bit integers

Table 2: Parameters used to generate secret session keys.

are the values generated for IPSec [13]. For the private keys (r_a, r_b) both parties select a randomly-generated 768-bit integer that must remain secret. To minimize the risks of an attacker discovering the private keys while they are useful, the keys are regenerated every two hours (this default value can be configured as required).

The client and server generate their public keys using their private key values. The client's public key is $K_a = g^{r_a} (mod\ p)$, while the server's is $K_b = g^{r_b} (mod\ p)$. These public keys are exchanged in the XCRYPT transfer described above. To generate the shared secret, the client computes $K_b{}^{r_a} (mod\ p)$, and the server computes $K_a{}^{r_b} (mod\ p)$. These two computations yield the same value, a number that must be kept private from other parties. Computing this shared secret is relatively expensive.

At this stage, email could be encrypted using the shared secret. However, for two reasons, there is another step in the process. As detailed below, ssmail is designed to avoid recalculating the shared secret unless absolutely necessary, which ensures that the encryption process is as fast as possible. We wish to be able to reuse the shared secret if the same client/server pair communicate again before either party's public key has changed. This is not a valid solution on its own. Re-using a key is an insecure practice: it increases the likelihood than an attacker could decrypt mail, particularly if a stream cipher is used in the encryption.

To avoid this problem, ssmail uses nonces (numbers used only once in the protocol) to ensure that a fresh private session key is generated for every exchange. The client and server exchange 80-bit randomly-selected integers. The shared secret and both nonces are passed through the Secure Hash Algorithm (SHA-1) [12] to make a 160-bit key. Messages sent from client to server use the first 80 bits as the key, and the last 80 bits are used as the key for the server to client messages.

The Encryption Algorithm

The 80-bit secret keys maintained by the client and server can be used in conjunction with any number of cryptographic algorithms. Originally, ssmail supported SOBER, a stream cipher developed by Greg Rose [15]. SOBER was specifically designed to support fast software encryption. This version of ssmail was never widely deployed, and so SOBER has been replaced. Instead, there are two algorithms supported:

- arrsyfor (compatible with RC4) [16]
- t32 (a faster and stronger derivative of the SOBER cipher) [14]

Other algorithms can be easily added, with the server suggesting the algorithm that best supports its needs.

Increasing Efficiency

Speed was a major concern when designing ssmail. Because encryption is taking place on mail servers that typically handle a large volume of mail, a slow cryptographic system could paralyse a server. The slowest part of our implementation is the Diffie-Hellman computation of the shared secret (that is, computing $K^r\ mod\ p$). To avoid recalculating this value for every ESMTP session, we use a cache. Every time a connection is made, each party stores the other party's public key and the shared secret in a cache. If this same machine is contacted later, the costly Diffie-Hellman calculation may not have to be performed again.

The format of the cache is as follows:

Header
Server's public and private keys
Time at which keys were generated
Body
machine 1: public key, shared secret
machine 2: public key, shared secret
...
machine n: public key, shared secret

Each time a connection is made, this cache is checked. First, the timestamp on this machine's own key pair is checked, and if it is too old (two hours by default), a new key pair is created and the cache is purged. Otherwise, if the other party's public key is found, then the shared secret is reused, saving an expensive recalculation. The shared secret is then passed through a secure hash computation with the nonces, but this calculation is fast. This makes computing a session key feasible for every connection.

If the public key is not found in the cache, then the shared secret is calculated with this new public key, then hashed to produce a session key. The public key and corresponding shared secret are stored in the cache for possible future use.

When a machine updates its private key, the shared secrets in the cache become invalid, as they were based on calculations performed with an outdated (different) key. The cache is simply cleared of all entries.

This caching method vastly increases the speed of encryption for a small network of machines that send mail mostly to each other (such as large corporate mail servers communicating with each other).

Performance

We performed two sets of tests on ssmail. Both tests evaluated sendmail 8.8.8 with ssmail extensions. The SSLeay library [18] version 0.9.0 generated random numbers to perform the Diffie-Hellman calculations and to perform the SHA-1 secure hash function.

The first tests were performed on ssmail with SOBER as the encryption algorithm. The test machine had a Pentium 100 MHz microprocessor and ran Linux 2.0.0. These results are based on sendmail receiving a plain text email of around 64 kilobytes

with the key cache already primed (that is, no Diffie-Hellman computation).

Tests on normal sendmail (without ssmail) showed that 63.9% of program time was spent in the `collect()` function, which reads command input from the network socket. With ssmail installed, this increased to 66%, with total running time increasing by less than 5%. The `collect()` function performed almost half of the 128,000 calls to `crypto_fgetc()`, the function that delivers decrypted text using the SOBER algorithm. Similar results were found for sending email as well.

These results may vary for mailers other than sendmail 8.8.8, as sendmail makes extensive use of single character input and output. Other mailers with different methods of text processing, such as line-based methods, may have different overheads.

It must be noted that the Diffie-Hellman computation will slow down mail processing. This computation is performed every two hours (depending on the configuration), and whenever a new machine is contacted. On the Linux machine, this computation took approximately 0.5 seconds of CPU time.

Another set of performance measurements were undertaken after implementing the two new cipher methods (arrsyfor and t32). In the following discussion, all measurements were done on a COMPAQ DeskPro (Pentium Pro) at 233 MHz, compiled with `gcc -O3` on Linux 2.0.33.

One of the authors' mail traffic for just over two weeks was analyzed to try to estimate the effect of caching on the Diffie-Hellman computation. Over a period of 372.6 hours, 1,322 email messages were received with a mean size of 14,671 bytes (median 2,950 bytes). Of these, 994 were delivered by the corporate mail hub machine. In the following analysis, we assume the worst possible case. For example, we assume that a new Diffie-Hellman key would have to be computed every two hours. (This could occur if messages arrived steadily.) Furthermore, we assume that none of the communications with other machines (other than the corporate mail hub) were able to use the cached information. This is unlikely, but possible if all other machines sent at most one message each per two-hour cache period.

Let us consider the effect of caching on the Diffie-Hellman calculations. In the case of the corporate mail hub, one calculation is done every two hours, giving 187 key calculations over the test period. This leaves 807 messages sent efficiently using the cache. We must also consider the messages from other machines (assumed to miss the cache), a total of 328. Out of 1322 messages, 515 required the Diffie-Hellman computation, while 808 did not. Thus, the cache saved over 60% of these computations. This is a conservative estimate, and the benefit to the mail hub machine itself could be expected to be much greater.

Each Diffie-Hellman computation took 0.3387 s. Amortized over all email messages, this was an average of 0.1317 s per email message. This is a similar order of magnitude to the total processing time for an average message. Using lightweight Diffie-Hellman key agreement and an aggressive caching strategy adds little overhead to mail processing.

Once the shared secret has been calculated or retrieved from the cache, the session keys must be derived, the encryption algorithm must be keyed independently for each direction, and the traffic must be encrypted. Ignoring the command dialog, the bulk of the encryption applies to the email message contents. Table 3 shows the CPU time in μs taken for the various steps of each of the currently supported encryption algorithms. (Note that t32 is still undergoing changes to make it more secure, and thus its performance is likely to change slightly in the near future.)

SOBER, as used in the earlier round of testing, was somewhat less efficient than arrsyfor, so the total running time of sendmail with the new version of ssmail is expected to increase by less than 3% (not counting the Diffie-Hellman computation).

Operation	arrsyfor	t32
key generation (SHA1)	22.12	22.12
two key setups	127.6	6.06
message encryption arrsyfor speed: 8.947 MB/s t32 speed: 12.09 MB/s	1640	 1213
total	1790	1241

Table 3: CPU time taken.

Related work

There have been a number of projects developed to protect email from eavesdroppers. Our own work was inspired by John Gilmore's Secure Wide Area Network (S/WAN) project [5]. S/WAN is designed to protect IP traffic by adding IPSec protocol support to personal computers. This differs from our work by providing network-layer encryption that can protect more kinds of traffic than just email transmissions. However, ssmail offers a simple and fast solution for email protection that can be easily installed. This is an alternative for systems that cannot handle the overhead of IPSec protocols.

Paul Hoffman has proposed an SMTP extension for privacy and authentication using SSL [7]. If authentication is not used, then his solution is very similar to ssmail. However, Hoffman's scheme has no support for the caching of the shared secret value, which leads to greater overhead from increased public-key calculations.

There have been a number of projects that proposed to embed encryption programs in mail user agents. For example, Paul Leyland and Piete Brooks

proposed a secure email project for the JANET network that would imbed PGP into mail user agents [9]. Putting encryption into the user agent rather than the transport agent requires more applications to be modified. Furthermore, they found that the PGP key servers would be unable to keep up with demand.

It is worth mentioning that there have been proposed extensions to SMTP for authentication. John Myers has put forth a system in which the client and server perform an authentication protocol exchange at the start of an SMTP session [11]. As ssmail is not intended to perform authentication, Myers' extension deals with a completely separate issue. It is possible that both SMTP extensions could be used by those systems that demand authentication and privacy and are prepared to handle the protocol overhead.

Limitations

While ssmail provides an effective solution for email privacy, it is has some limitations. Due to the multi-process nature of sendmail, the cache used to speed up key generation is stored on disk. The shared secret stored here is vulnerable to being retrieved and used to decrypt future email transmissions. Should this risk be considered too great, a trade-off in efficiency could be made. The disk cache can be turned off, resulting in a new shared secret for every exchange (but with the cost of the Diffie-Hellman computation). Note, however, that the email itself would also be exposed while stored on the compromised machine. We class this as an active attack, and hence outside the scope of our work.

Because SMTP transfers usually follow similar patterns, certain bytes generated by the encryption algorithm can be determined. For example, a "MAIL From:" command is followed by a "RCPT To:" command. Knowing this plaintext could provide enough information to allow an eavesdropper to determine such facts as the length of the recipient's email address. Any good encryption algorithm should minimize the information leaked from an encrypted session. However, this weakness is worth attention to ensure that the algorithm chosen does not fall prey to this threat.

Another limitation comes from the nature of store-and-forward transfer of email. Mail messages may pass through many servers en route to their destination. ssmail decrypts mail at each server before re-encrypting it for forwarding, so the mail remains unencrypted in the queue. As mail could be stored for an arbitrarily long time, this is a point of vulnerability. One way to reduce this threat is to use another program to encrypt mail during storage in the queue.

Future work

At the time of writing, ssmail is in a beta test version. There are a number of features that we intend to add or improve. We would like to replace the SSLeay random number generator with Peter Gutmann's package [6], which appears to be stronger. ssmail has already been extended to support t32 and RC4, an experience that indicates that porting other stream cipher algorithms should prove straightforward. Lastly, further analysis of traffic characteristics from a major mail hub would prove valuable for better estimating the saving from key caching.

Conclusions

We designed and implemented a fast, non-intrusive method for protecting large quantities of email against passive eavesdropping. We have supplied ESMTP with a key exchange command and added encryption algorithms to sendmail.

Our method has three main advantages:

- ssmail gives users privacy without requiring them to know how to use encryption packages correctly
- ssmail does not add excessive delay to email transmissions
- ssmail does not require widespread deployment of public key infrastructure

We believe that ssmail is a valuable tool for providing email privacy to a great number of users.

Availability

Contact Greg Rose <ggr@qualcomm.com> for information on availability of ssmail. We have implemented it as a patch to sendmail in order to prevent sendmail itself from being restricted by export regulations. At the time of writing, export licenses have been granted for recipients in five countries including the Czech Republic.

References

[1] B. Costales and E. Allman. *sendmail (Second Edition)*. O'Reilly & Associates (Sebastopol, CA), 1997.

[2] S. Deering and R. Minden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, Internet Engineering Task Force, December 1998.

[3] W. Diffie and M. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, 22 (1976), pp. 644-654.

[4] J. Galvin, S. Murphy, S. Crocker, and N. Freed. *Secure Multiparts for MIME*. RFC 1847, Internet Engineering Task Force, October 1995.

[5] J. Gilmore. *S/WAN: Securing the Internet against wiretapping*. http:/www.toad.com/gnu/swan.html.

[6] P. Gutmann. Software generation of practically strong random numbers. In *Proceedings of the Seventh USENIX Security Symposium*, San Antonio, TX, January 1998, pp. 243-257.

[7] P. Hoffman. *SMTP Service Extension for Secure SMTP over TLS*. RFC 2487, Internet Engineering Task Force, January 1999.

[8] J. Klensin, N. Freed, M. Rose, E. Stefferud and D. Crocker. *SMTP Service Extensions*. RFC

1869, Internet Engineering Task Force, November 1995.

[9] P. Leyland and P. Brooks. *Report of the UKERNA Secure Email Project*. http://www.cam.ac.uk.pgp.net/pgpnet/secemail/q4

[10] A. J. Menezes, P. C. van Oorschot and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press (Boca Raton, FL), 1997.

[11] J. Myers. *SMTP Service Extension for Authentication*. Internet Draft draft-myers-smtp-auth-12.txt, work in progress, Internet Engineering Task Force, November 1998.

[12] National Institute of Standards and Technology, NIST FIPS PUB 180-1, "Secure Hash Standard," U.S. Department of Commerce, April 1997.

[13] H. Orman. *The OAKLEY key determination protocol*. RFC 2412, Internet Engineering Task Force, November 1998.

[14] G. Rose, P. Hawkes. The T-class of SOBER Stream Ciphers. Unpublished manuscript. http://www.home.aone.net.au/ qualcomm

[15] G. Rose. A stream cipher based on linear feedback over $GF(2_8)$. In C. Boyd and E. Dawson, eds., ACISP'98 (Lecture Notes in Computer Science 1438), Springer Verlag, 1998.

[16] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C (Second Edition)*. John Wiley and Sons (New York, NY), 1996.

[17] A. Whitten and J. D. Tyger. *Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0*. In Proceedings of the Eighth USENIX Security Symposium, Washington, D.C., August 1999, pp. 169-183.

[18] E. Young. SSLeay Libraries Version 0.9.0. http://www.cryptsoft.com/ssleay/faq.html.

[19] P. Zimmerman. *The Official PGP User's Guide*. MIT Press (Cambridge, MA), 1995.

Author Information

Damian Bentley is currently studying self-modifying computer viruses for honours at the Australian National University. Previously Damian has worked for CSIRO and DSTO (Australian Scientific/Defence research organizations), and for QUALCOMM Australia on cryptography. More details about Damian and his work can be found at http://mehta.anu.edu.au/~bendchon, or he can be emailed at <bendchon@mehta.anu.edu.au>.

Greg Rose joined QUALCOMM in July 1996 as a senior staff engineer and manager, working on cryptography and authentication for the GlobalStar satellite communication project and the CDMA cellular phone system, and to set up the office of QUALCOMM Australia. He is enjoying work immensely. Greg is also Vice President of USENIX. He can be contacted by email at <ggr@qualcomm.com>, and his personal web page is http://people.qualcomm.com/ggr.

Tara Whalen is a researcher investigating network security at the Communications Research Centre Canada. Other research interests include wireless and mobile computing, human-computer interfaces, and social implications of computing. Tara holds an MMath in Computer Science from the University of Waterloo. She can be contacted at <tara.whalen@crc.ca>.

MJDLM: Majordomo based Distribution List Management

"It's not spam when WE send it"

Vincent D. Skahan, Jr. and Robert Katz – The Boeing Company

ABSTRACT

In early 1998, we were asked by Corporate Communications to develop a facility for providing a subscription based internal company mailing list capability that would permit Senior Executive management to send messages on an irregular basis from anywhere in the world.

Shortly thereafter, we were also asked to provide a one-use-only mechanism to send all 175,000 worldwide employees a (self-qualifying) message of where internally to report perceived spam in order to support corporate efforts to reduce incoming spam through technical means. Much to our surprise, delivering this message was a technical non-event.

The success of these efforts led Corporate Communications to request a more general system for permitting multiple mailings to targeted audiences, with a goal of completely eliminating the paper-based communications Management Information Bulletin systems, hopefully at considerable cost savings.

Given time and budget constraints, we chose to base our solution on majordomo.

This paper describes a scalable method for handling deliveries to multiple majordomo mailing lists with a minimum of administration. Ancillary issues such as sender authentication, message constraints, bounces, mail replies, and mailing list recipient management are also described.

This system is in use literally daily in Boeing and has easily supported lists as large as 150,000 recipients – drawn from custom SQL queries of the company's employee database. Conservative estimates of the savings in moving to a fully electronic communications mechanism exceed $1,000,000 per year with cycle time that has improved from several days to only a few hours.

Introduction

Beginning in January 1998, we were asked by Senior Executive management to provide a subscription based internal company mailing list that would be used for sending messages on an irregular basis.

Since the messages were coming from Senior Executives, there was a feeling that (of course) everyone in the company would ultimately choose to subscribe, making the potential audience 175,000 employees.

The only way to implement what was requested in the required time frame was to try to leverage our extensive experience with majordomo[1] and to develop a custom set of tools and processes that could be wrapped around majordomo. But scale was an obvious concern.

While majordomo has been in use within Boeing since 1992, usage profiles tended to be many reasonably small volume, small size (under 1000 recipient) lists. An Internet search of the majordomo-workers list archives appeared to show that majordomo had been proven only to approximately 25,000 recipients even when adding such optimizations as bulk-mailer.

We were talking about 'moving the decimal point' in terms of scale, and doing so for the highest levels of Senior Management. Things seemed bleak.

A Fortuitous Request

Shortly thereafter, we were asked to actually send a special "help us stop incoming spam" message to all e-mail enabled employees. We chose to use this one-time-only message as a test of how majordomo scaled within our existing hub-based electronic mail environment, which at that time supported well over 170,000 potential recipients.

(To say the least, all involved on the technical side quite enjoyed the irony of the message text.)

A team of internal majordomo, electronic mail, network infrastructure, and mail hub experts was put together to develop a plan to do a staged test with the appropriate instrumentation to see where things would begin to "brown out."

In addition, we set up the majordomo lists as exploder lists made up of 2000 recipient sublists, so that when things eventually failed, we could resend the messages more effectively to the victims.

Much to our surprise, even the largest (90,000 user) list was delivered with no problems, with the messages leaving the majordomo system in 5.8 minutes and getting through the mail hub (where readdressing to explicit smtp address occurs) and out to the variety

of destination systems (Exchange, UNIX SMTP, CC:Mail, etc.) in under 2.5 hours (see Figure 1).

A project was born.

Project Background

The Distributed List Management Service (DLMS) project was launched in mid-1998 to accomplish the following objectives:

- Support the requirements of Senior Executives to quickly, effectively, and completely disseminate information to ad hoc subsets of Company employees.
- Have the service work through the existing Public Relations and Communications Focals
- Adhere to the concept of alerting with E-mail and informing via the Web.
- Transition company-wide communications from a paper mail to an E-mail/Web basis
- Give the message author total control over (and responsibility for) message content.

A multi-disciplinary (8-person) team was assembled from database, mailing list, mail systems, communications and Public Relations specialists. The project had Vice Presidential sponsorship and produced the first working list within six weeks.

A message was sent to that list (10,331 recipients) on July 17, 1998. The Service was put in full production on April 24, 1999 with 65 active mailing lists.

MJDLM Requirements

There was considerable negotiation regarding what we would and would not support, and what constraints would be defined for the system and users.

Types of Lists

We envisioned two types of lists, temporary (one use), and permanent (more than one use). Temporary lists were intended to be disabled one week after use.

Message and List Limitations

Messages would be limited to majordomo's default 40,000 character limit in ASCII only.

Users were (very strongly) requested to limit their messages to brief pointers to web pages, in order to limit network and system load as much as possible.

Authors were fully and completely responsible for their content.

We agreed not to try to stop any message after the approved sender hit the "send" key.

All lists would be set up within majordomo as "restrict_post," "private," with moderation and advertising off.

Assembly of the Recipient List

It was decided that recipient lists would be created by SQL queries of the corporate personnel database, and would be updated weekly. Majordomo would then blindly believe that the SQL query and resulting employee e-mail addresses it returned from the corporate databases were complete and accurate.

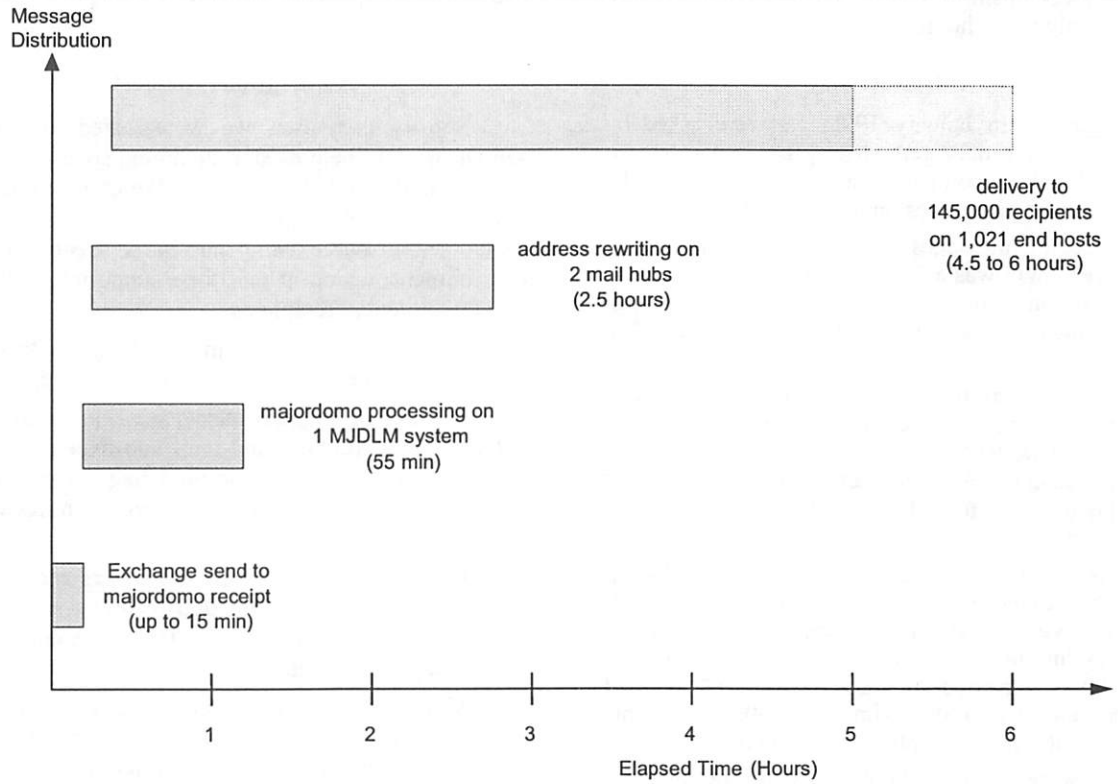


Figure 1: Typical MJDLM message delivery flow and latency.

Each REXX script (see Figure 2) would sort the recipients by the host portion of their Email address and produce a list of recipients which is suitable to FTP to the Majordomo Server. A separate REXX script for each list would also produce the number of employees with no E-mail who would otherwise fit the criteria.

Prior to actually using the data, a "sanity check" program would be run in advance to ensure proper address format, uniqueness, and continuity versus the previous week's list.

The costly address rewriting from "stable" to "explicit" e-mail address would take place on the corporate mail hubs to effectively serve as a throttle to prevent system and network "brown out."

Leveraging the pre-existing multi-hub architecture and associated load-levelling removes the need for special sendmail, majordomo or operating system tuning [3,5] on the MJDLM systems themselves that would have been necessary if the Majordomo system had to do the actual name resolution and delivery itself.

In addition, it permitted us to use whatever existing (weak) hardware we could acquire internally without the delay and stress of having to procure additional hardware for an unproven implementation.

Security and Network Considerations

Any message sent to more than 30,000 recipients would automatically notify those in charge of the mail hub(s) and DLMS agents.

Headers would be rewritten only by majordomo's resend program.

All MJDLM lists would have a restrict-post file of authorized senders and an explicit Reply-To: field.

Only the majordomo userid would be permitted to run the MJDLM program and modify list data.

The system needed to function in production on a 7 day x 24-hour basis.

Errors, Traceability and Message Archiving

Each message sent would be archived for a maximum of one year, but there was no requirement to permit retrieval of archived messages (majordomo's archive2.pl or the like).

Bounced messages would be the responsibility of the List-owners, with support from messaging specialists as needed.

Currently (by policy), all lists are owned by MJDLM project personnel rather than the various corporate communications focals, with one mailbox getting all the incoming bounces from all lists.

Bounces in the form of undeliverables, mail failures, host unknown, user unknown, warnings, and the like are collected in the list-owner's Mail subdirectory and managed by the procmail program via extremely simple recipes (see Figure 3).

These bounced messages help to verify certain recipient's complaints about not getting the message and wanting to. (There are also complaints about getting the message and not wanting to.)

Messages sent to the mailing list by unauthorized senders would be permitted to bounce for approval to the list owner, where they would be silently (by policy) ignored.

```
# To list all employees:

    BEMS_EMPL_ID        NE      ''      # Employee number exists and
    EMAIL_ADDR  NE      ''              # E-mail address exists and
    PEOPLE_TYPE IS      E               # Is a Boeing Employee and
    EMPL_STAT_CD        IS      A       # Is on Active Status

# To limit to Puget Sound, WA Engineering Managers add:

    TBC_LP_7            EQ      S       # Is in Puget Sound and
    ENTITY_ID           EQ      TBC     # Is in Heritage Boeing and
    PAY_CAT             EQ      M       # Is a Manager and

# Where specific Budget Organization names match the 1st character or
# 1st 3 characters as indicated for Engineering

    WHERE ((ORGN_STRUC EQ '4FT9')
OR (EDIT(TBC_BUDGET,'9') EQ 'B' OR 'E' OR 'U' OR 'R' OR 'N')
OR (EDIT(TBC_BUDGET,'999') EQ 'MAB')
OR (EDIT(TBC_BUDGET,'999') EQ 'MGB')
OR (EDIT(TBC_BUDGET,'999') EQ 'MHB')
OR (EDIT(TBC_BUDGET,'999') EQ 'MMB')
OR (EDIT(TBC_BUDGET,'999') EQ 'MTB'))
```

Figure 2: Example SQL Query Criteria for Targeted Lists.

Replies sent to the list instead of the Reply-to: address would be stopped by the restrict_post configuration, but the List-owner would manually forward it to the message sender as a courtesy.

How It Works From the User Perspective

Figure 4 shows a process data flow of list construction and handling, message sending and handling of returned mail.

The list is created after a form is filled out and approved by one of 13 company-wide communication focals. The criteria are researched (e.g., All Engineering Managers in Puget Sound, WA) and the SQL query developed to produce the recipients of the list.

Test messages from bonafide addresses are sent to verify the information on the form. When the list is ready, it is created using the MJDLM commands (see Appendix 1).

An E-mail letter is sent to all who are permitted to send to this list indicating the list name and address to send messages, the criteria, the number of recipients with and without E-mail, and orienting rules of use.

Lists are updated on Monday mornings, after weekly corporate database updates occur on the previous Friday evenings. Lists get disabled if there have been one-time messages sent to temporary lists or if the list becomes obsolete due to organizational makeovers.

```
:0
* ^Subject: Returned mail: User unknown
bounces.user-unknown
:0
* ^Subject: Returned mail: Host unknown
bounces.host-unknown
:0
* ^Subject: Returned mail: Local configuration error
bounces.cfgerror
:0
* ^Subject: Returned mail: Non delivery
bounces.nondelivery
:0
* ^Subject: Returned mail: Too many hops
bounces.hops
:0
* ^Subject: Undeliverable
bounces.undeliverable
:0
* ^Subject: Message status - undeliverable
bounces.undeliverable
:0
* ^Subject: unable to deliver
bounces.misc
:0
* ^Subject: Returned mail
bounces.misc
:0
* ^Subject: Non-delivery Report
bounces.delivery-reports
:0
* ^Subject: Delivery-Report
bounces.delivery-reports
:0
* ^Subject: Delivery Notification
bounces.delivery-reports
:0
* ^Subject: DELIVERY FAILURE
bounces.delivery-failure
:0
* ^Subject: Warning
bounces.warnings
:0
* ^Subject: Mail failure
bounces.mailfailure
```

Figure 3: Procmail Recipes.

The philosophy is to do the updating to all lists at once, rather than handle each explicit list individually. Other list handling is done on an exception basis.

When a message is sent to a mailing list, it goes out unmoderated. If the restrict-post addresses are not matched, it bounces to the list owner. As a courtesy, the sender is called to indicate this situation, since the sender may not necessarily be qualified to receive the message. Other bounces are ignored. Usually, the sender resends the message from the correct mailbox.

The major senders of weekday messages to all employees with E-mail are the Company electronic newspaper (daily) and the CEO's message of the week. Some other messages are timed to be released at a certain time of day for delivery purposes due to their media importance.

For the "all employees" list consisting of 144,561 recipients on June 30, 1999 there were 566 bounces in all categories yielding a return rate of 0.39% (see Figure 5). This is purely a function of company employee database and DNS accuracy.

Hardware Considerations

The MJDLM software and associated data runs on two Sun Sparcstation-5 systems with 128 MB RAM and one 2GB Hard Disk each.

One system serves as the primary server, with an MX record and associated majordomo and sendmail configurations to permit the backup to take over automatically if the primary production server is unavailable.

Both systems get full production monitoring and support on a 24x7x365 basis.

The actual MJDLM data is updated identically on each server via the MJDLM "update" functionality rather than updating the primary server and then synchronizing the backup server through other possible means such as rdist or mirror.

Since the throttle in the entire system is the time necessary to rewrite the addresses on the mail hubs, using slower than possible majordomo systems was not a problem, and served as a considerable cost savings since the hardware was in-house.

Centralizing mail delivery to occur by the mail hubs also permitted us to leverage the existing excellent logging and problem resolution capabilities already present there.

Why Majordomo?

We selected majordomo as the (hidden) list exploder subsystem mainly due to our existing experience with the product, our need to restrict postings to a very small set of author addresses, and the known behavior of how it permits easy header modifications (Reply-To: and the like).

We also had a pre-existing implementation in one small corner of the company that used a similar database query to majordomo model to routinely generate and update org-chart based distribution lists.

While this previous implementation was on a much larger number of lists (over 400) but far smaller

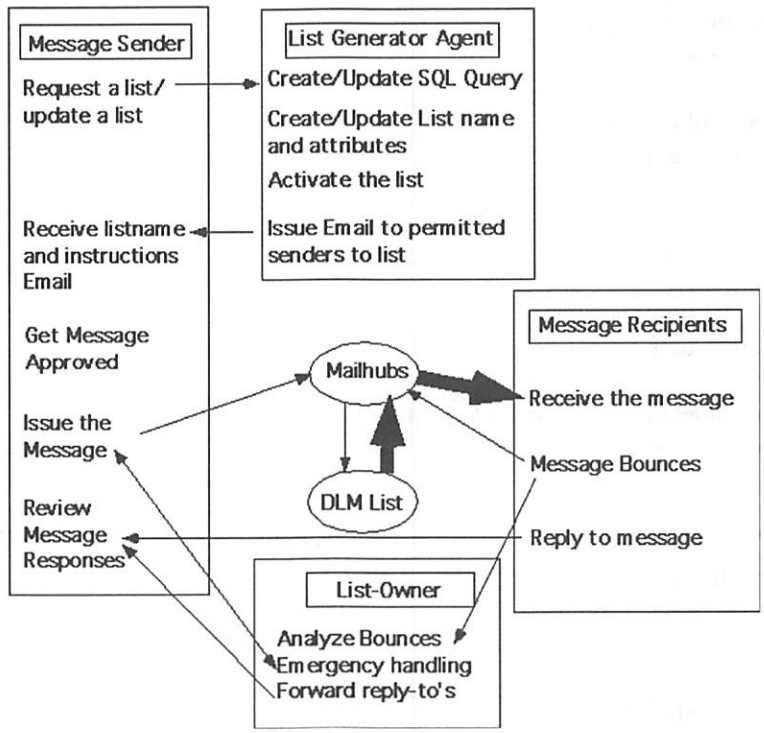


Figure 4: Basic Process and Message Flow.

number of recipients per list (under 1000) scale, we had five years of experience saying that the basic architecture was sound.

Lastly, the usual time and budget constraints prohibited acquiring, learning, and implementing any of the various commercial packages that claim to be able to support lists that could approach 200,000 recipients in size.

Custom Code Description and Risks

Our most important design point is that "*we will not modify majordomo in any way.*" This permits us to leverage the known behavior of majordomo without having to "roll our own."

This decision has caused us to turn down some user requests for features that are contrary to majordomo's implementation or general philosophy.

For example, users who tend to forget to send from approved mailboxes asked for a modification to return a "you did it wrong this way" message to them rather than a (hidden) bounce for approval to the list owner.

Internally, the MJDLM software is written in object oriented perl that uses a multilevel Data::Dumper structure containing all the information that needs to be stream edited into the per-list majordomo configuration files, or that needs to be saved, to permit adding, updating, disabling, and restoring a list (see Figure 6).

All installation-configurable parameters are present in external config files, with almost all routines in perl modules for reuse. There's little reusable content in the MJDLM perl program itself.

Lists are created on the majordomo side by the equivalent of sed on a template majordomo config file. This has a number of associated risks that we felt were acceptable for the foreseeable future:

- We're hard-wired to a particular version of majordomo. The major upgrade to majordomo 2.0 will cause a significant rewrite of much of the internal implementation.
- We must ensure that the config files we write are absolutely perfect, as we go around majordomo's existing "writeconfig" sanity checks.
- We have to get the template files correct right away due to poor inherent ability to change the template config and "make it so" on existing lists. We currently must disable/enable each list to cause a rewrite according to the current template.config.

All lists use the same template majordomo configuration file, although we support use of custom per-list configuration file templates if needed.

System Tuning

We have done nothing to tune sendmail or the operating system in any way, relying on sendmail 8.x and Solaris 2.6 to do the right things.

Number of recipients			144,561
No E-mail			45,860
Total That Matched Criteria			190,421
procmail filter files			
bounces.delivery-failure			15
bounces.delivery-reports			12
bounces.host-unknown			21
bounces.misc			36
bounces.slvma			9
bounces.undeliverable			416
bounces.user-unknown			12
bounces.warnings			0
list owner filtered Bounces Subtotal			521
Bounces to reply-to address			39
	delivery-failure	35	
	undeliverable	1	
	user-unknown	3	
other bounces to list-owner			6
	delivery-failure	2	
	user-unknown	4	
Total Number of Bounces			566
Total % of Bounces			0.3915%

Figure 5: Bounce Analysis for All Employees List June 30, 1999.

Since all the address rewriting of employee e-mail addresses from "stable public address" to "current real address" happens on the mail hubs, it really doesn't matter how fast the message clears the majordomo host as the bottleneck is the address rewriting on the mail hubs.

Majordomo is minimally tuned for even the largest lists. Lists under 2000 recipients are unaltered in any way. Lists over that size are configured as "exploder" lists consisting of 2000-recipient sublists.

While this tends to agree with some of the published documentation [5] regarding parallelizing delivery to large lists, in our particular case, the value originated from blind fear.

The value 2000 represents twice the largest majordomo list we had run in production, and approximately 10% of the size of the largest list we could find mentioned in the majordomo-workers archive.

The thought was that if a delivery "broke," we would be able to determine which sublists were hung in the queue, and manually re-send the message to just those recipients if needed. Fortunately, this hasn't happened yet.

Optimizing Delivery Time

While the majordomo system can run safely unoptimized, delivery time to the end users is an important success metric for the project.

We originally structured the SQL queries to generate output sorted by recipient employee messaging system unique ID number (optimized for the human, not the computer).

Since the employee database knows both the unique "key" for the employee, and the "value" of the e-mail address where that user ultimately reads their mail, it was suggested by the mail hub personnel to alter the SQL reports to sort based on the hostname the recipient mail really goes to.

This got us a 55% decrease in number of messages required to be delivered and indirectly a reduction in total delivery time (ala bulk mailer) by letting the mail hub deliver to the destination addresses in a streaming mode.

```
List::new (code admittedly derived from Perl5 Camel p298)

=item new()

This creates a List object with the contents initially
undefined. It is up to the calling program to populate
the 'pieces' of a List object.

=cut

%fields = (
     # the following items are specified by the users
     name              => undef,   # domo list name
     owner             => undef,   # domo owner e-mail address
     description       => undef,   # domo description string
     reply_to          => undef,   # follows domo reply_to values
     restrict_post     => [],      # e-mail of approved posters
     # the following items are calculated
     num_sublists      => undef,   # number of 'sub-lists' needed
     list_type         => undef,   # temporary or permanent
     isa_sublist       => undef,   # am I a sublist ?
     staged_recipients => undef,   # filename of last recipient list
     num_recipients    => undef,   # number of recipients
    );

sub new {
  my $that  = shift;
  my $class = ref($that) || $that;
  my $self  = { _permitted => \%fields, %fields, };
  bless $self, $class;
  return $self;
}
```

Figure 6: Perl Code Example.

A total of 10 MS-Exchange bridgeheads receive mail for 91% of the employees with E-mail addresses. Approximately 1,000 other systems receive the remaining 9% of the messages.

The lower number of end messages from the mail hubs also helps protect the network from overloading.

Use To Date

Between July 17, 1998 and June 30, 1999, there have been a total of 1278 messages sent to 55 lists, with 38 lists having yet to receive their first message. The average number of messages sent to all lists is 106.5 per month.

Of the 1278 messages sent, the three largest lists (currently averaging 149,802, 72,258, and 34,730 recipients each) have processed 155 or 12.13% of the messages sent while they generated 87.32% of the actual traffic (or 21,833,519 messages) of the total 25,003,407 discrete message deliveries to date (see Figure 7). Since the number of employees has been declining since the beginning of this year, this represents a lower bound value.

The top three lists together issued 1.6 messages per weekday for June 1999.

Web Data Interface

An independent adjunct to this project is the use of a daily updated internal Web site that provides a List Request or List Change form, the status of the Server (if down), the status of lists both existing and gestating, and a variety of database driven reports that map sender names to permitted sender Exchange Group mailbox names. Focals and their associated existing lists, and Reports for each list indicating the permitted sender and other attribute information for that list are also viewable.

This permits the information flow to be centralized and authoritative for project members, senders and interested parties. Much of this data is password protected at this time and disclosed only to Focals and project members.

Benefits

MJDLM provides a new capability to perform an ad hoc mailing list distribution based on a Company Employee database that is refreshed weekly. This permits targeted individual messages as well as message-series to be sent on a regular basis.

Annual cost savings are estimated to be $1,000,000/year in reduced or eliminated paper printing costs, with additional improvements in cycle time by permitting real-time creation and distribution of time-critical communications by using Internet Standard Time for message creation, approval, sending and distribution rather than internal or external "Snail Mail.'

Risks and Concerns

There is concern that as people get accustomed to the service, their expectations might start to exceed the infrastructure (or the recipients') capacity to absorb such broadcasts.

	Messages per Month	Messages per weekday
Jan-99	17	0.85
Feb-99	20	1.00
Mar-99	28	1.22
Apr-99	32	1.45
May-99	26	1.38
Jun-99	32	1.60

Large List Name	Average Size 6/30/99	Messages Sent by 6/30/99	Messages Received by 6/30/99	Estimated Bounces by 6/30/99
All Employees	149,802	141	21,122,131	82,700
All Commercial Airplanes	72,258	6	433,547	1,697
All A/C & Missile Systems	34,730	8	277,841	1,088
Subtotal for Large Lists		155	21,833,519	85,485
Total All Lists		1278	25,003,407	97,896
% of Large Lists to All Lists		12.13%	87.32%	

Figure 7: Frequency of Use For Large Lists (over 30,000 recipients).

Fortunately, the Company Focals who decide who can have a list created and what it can be used for have been judicious in their creation and use of "big" (over 30,000 recipient) lists and helpful in limiting their use of big lists to times outside Puget Sound prime-time.

The leading use of the MJDLM system in terms of number of delivered messages has been in presenting a new electronic "Boeing News Now" daily to all employees, containing short news briefs thought to be of interest to all employees, with associated pointers to the appropriate web pages. This service broadcasts five days a week at 9:00 p.m. Seattle time.

Especially with the creation of an all employees list, there were initial fears of mass resistance to being "spammed" by a daily news service. It turned out that relatively few wished to publicly express the desire to opt out of receiving these mailings. Those that had religious views (interestingly enough, which included much of the project team) were able to filter out the messages on their own.

At this time, however, *only unemployment or a mistake in the Company Employee database prevents a recipient from being included on the various targeted distribution lists*. This appears to be an unalterable political reality at this time.

Futures

Internally, as we are "overrun by success," we believe the one file per list database structure we're using internally will need to be updated to a multilevel DBM or LDAP configuration for scalability.

Our command-line interface is admittedly minimal, and enhancement with a GUI or web front end [2,4] is needed for usability.

Even with a relatively low bounce rate, we have encountered situations where real-time use of procmail to filter incoming bounces has caused system stability problems. We are considering filtering bounces in batch mode [3] rather in real-time via procmail or similar tools.

There are also plans to better integrate with the continuing evolution of Company databases over the next few years and to fully automate the current (largely manual) status-related web pages.

In the meantime, there will be continued study and fine-tuning of a mailing system that all levels of corporate management have embraced and are beginning to rely on as part of their daily business processes.

Acknowledgements

The authors wish to acknowledge the following people who influenced the quality and fine-tuning of MJDLM.

They include Mary Jo Shipley, Rick Willard, Laurie Thompson, Dean Richardson, Sean O'Sullivan, Paul Swortz, Don Meyer, Brenda Kern, Patrick Palmer, Denise Sirven, Jim Crozier, Ron Hanson, Karen Burt, and Don Fitts.

Author Information

Vince Skahan <vince.skahan@boeing.com> is a System Design & Integration Specialist at the Boeing Company. He holds a B.S. in Chemical Engineering from Drexel University, and has been doing large scale unix administration since 1987. He's run majordomo since immediately after learning about it at the 1992 LISA conference, and also admits to being the original author of the majordomo FAQ.

Robert Katz <robert.katz@boeing.com> is a System Design & Integration Specialist at the Boeing Company. He holds an M.S. (Math) degree from Polytechnic Institute of Brooklyn. His technical interests include Large Scale E-mail communication, UNIX, Perl, Transaction Processing Monitors and CORBA. He teaches UNIX: Introduction, Shell Programming, and System Administration at Highline Community College in Des Moines, WA in his spare evenings.

References

[1] Chapman, B., "Majordomo – How I Manage 17 Mailing Lists Without Answering '-request' Mail," *Proceedings of the Sixth Systems Administration Conference (LISA VI)*.

[2] Viega, J., et al., Mailman: The GNU Mailing List Manager, http://www.usenix.org/publications/library/proceedings/lisa98/viega.html.

[3] Rose Chalup, S., et al., "Drinking from the Fire(walls) Hose: Another Approach to Very Large Mailing Lists," http://www.usenix.org/publications/library/proceedings/lisa98/chalup.html.

[4] Houle, B., "MajorCool: A Web Interface To Majordomo," http://www.usenix.org/publications/library/proceedings/lisa96/bhoule.html.

[5] Kolstad, R., "Tuning Sendmail for Large Mailing Lists," http://www.usenix.org/publications/library/proceedings/lisa97/21.kolstad.html.

APPENDIX 1: Example List Creation Transcript.

```
# mjdlm --help

 Version 2.09 usage:  mjdlm [-options] [-list LISTNAME] [-misc_actions]
 The following options operate on all lists, unless a particular list
 was specified (which restricts the action to the specified list only):
 ------------------------------------------------------------------------
  -add             = add all lists needing adding
  -update          = update all lists needing updating
  (none specified) = do all required additions/deletions/updates

 Miscellaneous options include:
 ------------------------------
  -help      = give help message
  -debug     = run in debug (but not execute) mode
  -verbose   = verbose output
  -prompt    = prompt user (used to create a new control file)
  -show      = show control file for a particular list
  -showlists = show what lists currently exist

 The following options REQUIRE a list to be specified:
 -----------------------------------------------------
  -temporary = convert the list to temporary status
  -permanent = convert the list to permanent status
  -restore   = restore the list from the 'trash'
  -rename    = rename a list (and its sublists)
  -disable   = disable a list (and its sublists)

# mjdlm -prompt

     Enter the desired list name.  Majordomo lists need to be
     lowercase, with no embedded spaces or ''funny'' characters other
     than [a-z], [0-9], and perhaps "+", "_", and "-"

list name [] : mytestlist

    Enter the desired reply_to configuration.  The choices are
       $SENDER  (reply to the message sender)
       "list"    (reply to the whole list)
       or enter the full e-mail address of who you want to receive replies

list reply_to setting [$SENDER] : <return>

    Enter a 50-char-max description of the list

list description [] : This is a test list

    Enter the desired reply_to configuration.  The choices are
       $SENDER  (reply to the message sender)
       "list"    (reply to the whole list)
       or enter the full e-mail address of who you want to receive replies

list owner [owner1@host.domain] : <return>

    Enter the full e-mail address(es) that can post a message to the list.
    If there is more than one person, you will get the chance to answer
    individually.  Just hit 'return' after the last address was entered.

list poster e-mail address (return to 'break out') [] : dude1@hostname.domain
list poster e-mail address (return to 'break out') [] : dude2@hostname.domain

list poster e-mail address (return to 'break out') [] : <return>

    Enter the type of list.  Choices are:
       permanent = the list stays forever
       temporary = the list 'expires' a few days after its use
```

```
enter list type (permanent|temporary) [permanent] : <return>

# mjdlm -add -list mytestlist
              MJDLM version 2.09
              ------------------

 found mytestlist      in  control staging         = create the list

....creating list "mytestlist"....

# mjdlm -show -list mytestlist
   name              = mytestlist
   owner             = owner1@hostname.domain
   description       = This is a test list
   reply_to          = $SENDER
   restrict_post     = dude1@hostname.domain dude2@hostname.domain
   num_sublists      = 0
   list_type         = permanent
   isa_sublist       = 0
   staged_recipients = mytestlist.permanent.19990708
   num_recipients    = 1234
```

Example sendmail aliases

```
# archived list => mytestlist
owner-mytestlist:    owner1@hostname.domain
mytestlist-owner:    owner1@hostname.domain
mytestlist-approval: owner1@hostname.domain
mytestlist:    "|/home/majordom/wrapper.dlm resend -l mytestlist \
                  mytestlist-list"
mytestlist-list:  :include:/home/majordom/mjdlm_data/listdir/mytestlist, \
                  mytestlist-archive
mytestlist-request: "|/home/majordom/wrapper.dlm majordomo \
                  -l mytestlist"
mytestlist-archive: "|/home/majordom/wrapper.dlm archive2.pl -f \
      /home/majordom/mjdlm_data/listdir/mytestlist.arch/mytestlist -m -a"
```

RedAlert: A Scalable System for Application Monitoring

Eric Sorenson – Explosive Networking
Strata Rose Chalup – VirtualNet

ABSTRACT

RedAlert is a complete application monitoring system which consists of a stateful server daemon and extensible Perl client API. Almost any IP-protocol service is a candidate for RedAlert monitoring: the clients determine what error condition they have discovered, convert that information into a standard message format, and transmit the Alert to the server.

RedAlert therefore will easily plug in to existing script-based monitoring environments, providing greatly increased functionality for a minimal investment in configuration time. This functionality includes volume tracking, interval sampling, threshold-based notifications, and reporting mechanisms which include pager, electronic mail, and SNMP traps.

We have chosen to focus on email monitoring, specifically postmaster bounce mail, for the scope of this paper. Bounce mail is both ubiquitous and complicated, making it ideal for RedAlert monitoring.

View from 25,000 Feet

RedAlert is an extensible, easy-to-use client/server framework written in object-oriented Perl. It comes with a couple of sample client programs and classes, a generalized API to create new kinds of clients, and a full-featured server which supports several types of threshold monitoring and notification channels.

The goals in its design were:

- easy integration into existing monitoring
- extendability on the client side
- the ability to catch failure modes which slip by traditional network monitoring systems undetected

True to its client/server nature, RedAlert is composed of two parts: the monitoring daemon and a set of clients which report to it. This enables a RedAlert installation to aggregate information about specific types of occurrences and trigger events based on administrator-configured thresholds.

Some of the highlights of the implementation:

1. Flexible connection model. RedAlert uses Perl's Data::Dumper [1] and TCP connections to pass objects across the network. The connection model doesn't care about an Alert's content, just its "well-formedness," which promotes extensibility.
2. Object oriented. RedAlert was built from the ground up using OO methodologies. If you have home-grown Perl scripts for automation, you'll find it easy to extend RedAlert's monitoring capabilities to your site. For example, given a Perl program which connects to a web server and checks for a known-good Content-length: header, it's ten additional lines of code to send a RedAlert warnings if any of the stages of the connection process fail.
3. SNMP notifications. RedAlert has several methods of triggering SNMP traps. Browsing support is coming, but is not implemented at the time of this writing. RedAlert has been allocated a registered branch of the 3Com Enterprise MIB. You are in no way obligated to download the 3Com MIB or use 3Com equipment. The important part is that notifications are tagged with unique registered ObjectID's so you won't have any conflicts with your existing SNMP setup.

Origins of RedAlert

Our system originally came about as a response to a site-specific need. The authors were coworkers at the site of a client who was building an Internet Access Service using some Netscape products, including Netscape Messaging Server. A few months into the beta test phase, we realized we were being overwhelmed by the amount of bounce mail the NMS' would generate. Load tests brought this problem into sharp relief: after 13,000 "user over quota" bounces from a test gone haywire brought down the operations mail server, we decided it had gone far enough.

We sensed the need for an intelligent front end to the postmaster mailbox. This front end would keep track of the types and quantities of bounces it received and determine whether they represented a "blip," a trend indicating a more serious problem, or an error which required immediate sysadmin attention. One "user not found" error probably isn't serious; thirty of them to the same user in the span of a few minutes, on the other hand, might indicate a mailbomb in progress. The client/server model lent itself to a polymorphic

network of application monitoring programs, with the client scripts communicating their results to a server which could make decisions on what a given error means.

The Status Mail Deluge

Many service daemons are equipped to report errors or unusual conditions via email. In the halcyon days where every user was also her or his own sysadmin, this was a friendly and useful way to report errors. The information would appear in your mailbox, perhaps even *biff*(1)'d across your screen.

With a typical crop of cron jobs, license monitors, network service daemons, and user email, even a small cluster of several to a dozen machines can generate a healthy quantity of email over the course of several days or a weekend. A mid-size engineering firm of two to three thousand employees may have several hundred servers and generate as much email in a day as our small cluster would in a month.

User service clusters, such as those deployed by Internet Service Providers, typically run a number of email-noisy security tools in addition to the normal utilities. Their volume, however, is dwarfed by that of postmaster mail. Running electronic mail services on an ISP scale can literally result in megabytes of postmaster mail per day.

From "Needs Practice" to "Best Practices"

In many small shops there is something of a laissez faire attitude towards email status notifications: "If something goes wrong enough, the users will tell us."[1] At the other end of the spectrum are the shops which have everything sorted, scripted, and routed. In ascending order of utility, here are the methods currently employed by most sites:

1. Check each server individually "when you get around to it." This is usually shorthand for "we will check postmaster/root mail when something breaks."
2. Define aliases on each host to funnel mail to a central collection point, i.e., alias root@thishost to root@mailhub. Read by hand in between fire-fighting.
3. Implement the centralized funneling, and add some form of regular expression filtering to sort mail into different files/folders. Procmail is a typical method, followed closely by the filtering options native to "whatever mailer the lead sysadmin prefers."
   At some sites, the processing is delayed rather than real-time, and the filtering is invoked regularly on the common inbox to perform the sorting. Read between fire-fighting, or have a junior team member keep an eye on the folders every day or two.
4. Implement item 3, and add scripting. The scripting seems to take one of two forms, and it is unusual (though praiseworthy) to see both employed at the same site.
   - Add an extra step or two to the filtering, sending particular messages to a team member's inbox or to the email input of a trouble-ticket system such as Remedy, RT, req or Scopus.
   - Add one or more cron jobs that rotate the types of folders (e.g., host-not-responding, user-not-found, etc) daily and watch the size of the folders between rotation. If a particular file gets larger than a site-specific "typical" size, notify via email to a pager gateway or trouble ticket system.

In practice, this means that site policy will usually be extreme in one direction or the other. Some sites will save everything, but practice "file and forget" or "dig through the attic in case of trouble." Other sites will turn off postmaster copies of bounces, set logging options down to "critical only" for servers, and flood some unlucky soul's pager or regular email box with everything that comes through.

Putting the "State" back in "State of the Art"

The critical piece which all of the above lack is an "intelligent" collection node. To qualify, a node should receive the incoming messages and be able to make connections between them based on content: their origin, destination, meaning, overall number, frequency during an interval, and so on. A solution relying on scattered individual files and filter-triggered scripts doesn't count.

All these sets of stateless, event-driven scripts are still the equivalent of counting elephants in a field with a fiberglass pole. You walk with the pole held out beside you, and every now and then it deflects on something, bends back, and smacks you in the face. You then page your postmaster with an "elephant spotted in Field Seven" message. It's up to the postmaster to keep track of how many pages came in that day, and which fields are rife with elephants.

Enter RedAlert's stateful server daemon: keeping tabs on troublesome elephants day and night. The client API allows for extensibility and greater sophistication in types of clients monitored, and is (in our humble opinions) a very useful contribution. Fundamentally, though, the server daemon serves as the keystone to the arch of intelligent status mail processing, transforming it from a pile of semi-organized building blocks into a definite structure.

For the rest of this paper, we'll walk through building a RedAlert client and server configuration to monitor a typical application, namely, bounce messages generated by the Sendmail daemon.

[1] This meshes particularly badly with what Elizabeth Zwicky calls "systems administration via psychic powers" but can be a useful predictor of an individual shop's normal uptime/downtime ratio.

RedAlert Systems Architecture

Since RedAlert can be deployed in a centralized or distributed fashion, you must decide how you wish to aggregate the alerts. This requires identification of your goals for the monitoring of your mail system.

Obviously if your site is a high-volume site you will have different goals than if you are running a small site. Individual local hosts or particular remote destinations may be of specific importance to you, and you may wish to have someone paged if "host not found" bounces begin popping up for that site. On the other hand, you may simply wish to have statistics recorded on the "miss" rate of user addressed email.

If you have deployed load-balanced banks of mail servers, you will probably wish to have a separate RedAlert server instance aggregating traffic for each bank. The prioritization of alerts for the banks may depend largely on upon whether your load-balancing solution can "busy out" an unresponsive server or if it must be done by hand.

Decide where to aggregate

You can either aggregate the information yourself by redirecting postmaster mail to your mailhub, or you can make the service machines do the work and use an SNMP collection tool for historical data and trend analysis. We recommend the latter in most cases, as it localizes and distributes the overhead of running the RedAlert service.

The approach you choose will be determined by your desired handling of the actual postmaster mail. If you are operating under a "notice, then delete" policy, it is of course more efficient to throw away the mail right on the originating server. Errors of a type which should be saved can presumably be redirected to a central server by the same filtering approach which you are using to invoke the RedAlert client.

Option 1) Centralize all mail to a mailhub, run it on the inflows, snmp monitor mailhub.

Option 2) Run it locally on the servers, (optionally routing a copy of the message to a central collection point for archiving), snmp monitor each server (which you're probably doing anyway).

Option 3) Run a distributed setup, with a config master for each class of service machine, aggregating via option 1 or option 2, or an option 3 recursive of "mailhub for class."

Decide What To Throw Away

Types of postmaster mail which a large site might wish to track but throw away include "hostname not found" and "user not found" originating at remote sites. It can be beneficial to know the numbers and frequency of these sorts of errors, but there is rarely information in them useful to maintaining your mail servers.

For example, an unusually large spike in "hostname not found" is worthy of an alert, as it could signal the demise of a DNS server on which that mail machine depends, the useful information is really in the fact that the spike occurred, not the content of the individual messages.

Typical Postmaster Alerts

Many of these are in the class of "Gee, if anyone actually read postmaster mail at our site, we would have seen this when it came through." This is exactly why we came up with the idea of RedAlert – because so few sites actually examine postmaster mail in a timely fashion.

We will use sendmail as an example, given its widespread acceptance across many types of installations. In our notification examples, it is arguable that any condition severe enough to generate an SNMP trap might also be a condition where you would want to page someone. However we will assume that the SNMP monitoring system has its own rules for paging, and that some trap-only notifications might result in pages. There are some things that are sufficiently bad that we would want to page anyway and risk double-paging some poor sod at 3am.

Please note that our examples do not constitute a comprehensive list of any and all errors, nor are the examples ranked in any particular order, most especially including likelihood or severity.

Severe MTA Error

Most types of 451 errors should cause system staff to be paged. Of course, some of them may indicate an OS-level problem of a severity that would preclude the operation of RedAlert as well. A prime example is "451 %s: cannot fork", which we'll use to build up our RedAlert configurations. See Table 1.

Performance Related Conditions

These are things which are useful to log or trap and later correlate against system and network load

Severe MTA Errors		
Error	Threshold	Notification
451 %s: cannot fork	0	Trap, Page, Log
452 Error writing control file %s	N per interval	Trap, Page, Log
452 Out of disk space for temp file	N per interval	Trap, Page, Log
451 %s: lost child	N per interval	Trap, Log

Table 1: Severe MTA errors.

data. This will let you see some cause and effect linkages more clearly than random poking. Many of these errors will never be seen at small or well-scaled sites. Tracking the remote host involved in timeouts will allow you to set your mailer timeouts to handle certain destinations which can be notoriously slow. A very high-volume site may wish to use mailertables to segregate this traffic to dedicated mail routers with unreasonably high timeout values. See Table 2.

Potentially Security Related Errors

Some mailer errors are most often seen in response to root-kit style cracking attempts. These in particular often involve strange terminations in the mail.local phase. Generating a trap for these errors gives NOC staff the ability to investigate in real-time.

Others of this category of error can be generated in response to spammers attempting to use the MTA for spam dumping. Logging "452 Too many recipients" and correlating against RADIUS authentication logs may enable ISP abuse desk staff to identify spammers. Some of the spam dumping programs out there are also poorly written and generate bad SMTP.

Some of the threshold values in our table might be better off with "N per interval" or even "Y rate of increase" instead of using single-Alert triggers. and which should just be "after some small number." This will depend largely on just how huge your site is and how peculiar your mail clients may be. Sites running cc:mail or Exchange gateways for users who love forwarding and love attachments may see these kinds kinds of errors on a fairly routine basis, for example.

Deploy Clients

Create clients for the various individual scripts or add cases in your existing filtering script for routing postmaster (root, daemon, cron, etc) mail. Substitute or add a RedAlert client script with the appropriate arguments and push the configuration files and modified filtering scripts out to the clients.

Performance Related Conditions		
Error	Threshold	Notification
451 open timeout on %s	N per interval	Trap, Log
451 reply: read error from %s	N per interval	Trap, Log
451 timeout waiting for input during message collect	N per interval	Trap, Log
452 Insufficient disk space; try again later	N per interval	Trap, Log

Table 2: Performance related conditions.

Potentially Security Related		
Error	Threshold	Notification
452 Too many recipients	0	Trap, Log
500 Bad usage	0	Trap, Log
051 WARNING: writable directory %s	0	Trap, Log
451 %s: died on signal %d	0	Trap, Log
550 Access denied		
550 Address %s is unsafe for mailing to programs	0	Trap, Log
550 User %s@%s doesn't have a valid shell for mailing to files	0	Trap, Log
500 Parameter required	0	Trap, Log
500 smtp: unknown code %d	0	Trap, Log
501 Syntax error in parameters scanning "%s"	0	Trap, Log
501 %s parameter unrecognized	0	Trap, Log
501 %s requires domain address	0	Trap, Log
501 Unknown BODY type %s	0	Trap, Log
502 Sorry, we do not allow this operation	0	Trap, Log
503 Nested MAIL command: MAIL %s	0	Trap, Log
503 Polite people say HELO first	0	Trap, Log

Table 3: Potential security errors.

While the client information could all be included in one master file, we recommend splitting it into service-specific and/or class-specific config files. Thus a mistake in editing your squid_proxy.conf file would not affect your smtp-in.conf, etc. This approach arguably makes deployment easier, as a large site could build up a library of different client modules to reuse and recombine for new situations.

The Configuration Files

Now that we know what we're looking for, it's time to start figuring out how to find it. "Finding it" in RedAlert's case means setting up your client and server configuration files in a consistent, complete, and logical manner.

RedAlert uses simple, text-based configuration files for both clients and the server. As mentioned above, it's generally better to keep a client's configuration pared down to only those Categories of alert which that client will be expected to handle; you win both in ease of use and execution speed – less to parse means less memory and time used in parsing.

Our configuration files look similar to a Windows .INI file. They contain section names in square brackets that identify the Category we're handling, and then list "name = value" pairs for relevant variables for that section. Using AppConfig, a Perl module by Andy Wardley [3], allows us a great deal of flexibility with config file parsing, so the niggling typos which plague primitive parsers aren't a problem.

The configuration file is divided into three main sections.

1. Global. The global section contains overall config information such as port number, system-wide defaults, etc.
2. Panic. The "panic" section contains configuration information for RedAlert to report errors on itself. Problems parsing an incoming client messages, failure to contact the server's RedAlert port, or even debugging strings can be sent out with a Panic, typically an email message sent to a host of last resort.
3. Category-specific. The rest of the file's sections contain the site-configurable settings for types of events, notification procedures, etc. These sections on the client side only contain a template for the Alert to send; on the server, they set up monitoring threshold and notification parameters.

Please note: It's very important to keep your Categories consistent between the clients' configuration files and the server's. An incoming Alert's Category determines which threshold will be incremented and processed. RedAlert tries to do something reasonable with unfamiliar Categories, but "reasonable" means sending a Panic message saying it didn't know what to do with the Alert. Receiving a large number of Panic messages due to a misconfigured client will dilute the legitimate Panics... in a truly degenerate situation, you might even be forced to run a meta-RedAlert server to filter incoming Panic messages!

We'll build up sample client and server configurations for monitoring our sendmail alerts as we discuss each component in turn.

Crafting a RedAlert Client

The first (and trickiest) part of making a RedAlert client is figuring out exactly what it is you're looking for. In the case of our sample client for parsing Sendmail bounces (hereafter called sendmail_client.pl), we poked through the sendmail binary with *strings*(1) and came up with appropriate regular expressions to match the various kinds of bounces the program can generate.

We then created a simple sendmail_client.conf, with templates for each of the different Categories of bounce we might receive. In the interest of brevity, this client example will only focus on one specific error from the tables above: "Cannot fork". This is a pretty serious error condition, indicating that sendmail had used up its allotment of child process id's, or (worse) that the machine's process table was full. We'll want to extract the "%s" substitution from the message and use it to fill in our Template; since we're dealing with sendmail here, the Message-Id of the bounce will come in handy too. So, our sendmail_client.conf looks like the code in Listing 1a.

The methodology you use to create and configure your own clients will depend upon the kinds of error strings your application produces – in the case of a Squid web-proxy monitor, as another example, the error conditions are indicated by known-bad responses

```
[global]
port = 7200
debug = 1
servername = alerthost.ops.domain.com
lastresort = redalert-admin@ops.domain.com

## Types of email bounces
[sendmail_cannot-fork]
template = "Sendmail couldn't fork doing _AAA! Message-ID: _BBB"
```

Listing 1a: sendmail_client.conf.

from the server (a 404 or 500 error to a page which ought to be there or a CGI which should be working), or by a failure in the various stages of establishing the TCP connection to the proxy's port. This level of monitoring is what makes RedAlert special – no matter how expensive the network monitoring solution in use at sites we've seen, it always seems to miss certain failure modes that always end up biting you in an uncomfortable place.[2] With RedAlert, however, you can catch errors specific to your services, plus adapt your monitoring routines to the new and interesting failures that seem to creep in with a new code revision or configuration change.

To promote reusability, we created a new subclass of RedAlert::Client called Sendmail. This translates into a file under the RedAlert library directory: RedAlert/Client/Sendmail.pm. To avoid losing sight of the more typical cases where you're integrating RedAlert client functions into already-existing programs or writing a simpler frontend that won't involve package creation, we'll focus more on the sendmail_client.pl interface and the Client API, delving into the new subclass' methods only when necessary. Consider the following code segments from sendmail_client.pl shown in Listing 1b.

The first two lines pre-load our module, then instantiate a new RedAlert::Client::Sendmail object. The "new" constructor method inherits attributes from both its parent classes, the base RedAlert class and the Client subclass; see Listing 1c.

This line calls the Configure method to parse our config file, loading up the categories and templates into which this bounce might fall. Configure is actually a RedAlert::Client method which we have inherited; see Listing 1d.

This is our only chunk of real client-specific code. Since we're destined to be passed an email, we can use Mail::Internet's builtin facility to create a new Mail object by reading from stdin, the odd-shaped [<>] construct in like 10. We use some handy methods in the Mail module to create references to Mail::Header and Mail::Body objects from the incoming email in lines 6 and 7, and pass these as arguments to the Parse method of our Alert object.

Here we'll have to descend into Sendmail.pm for a bit, to follow the Parse method as it fills in our Template with appropriate substitutions for the placeholders defined there. See Listing 2a for some code fragments from Sendmail.pm.

Line 2 might seem puzzling at first, but it starts to make sense with the knowledge that calling a method with the arrow operator, as in "$alert->Parse", passes the object's reference as the parameter to the method. So when we "shift" above, we're assigning the the first element of the implied @_ to $self. This gives us a local copy of the object to manipulate. The next two elements of @_ are references to the Mail::Headers and Mail::Body objects passed to us by the sendmail_client.pl; see Listing 2b.

Here's the meat of the bounce processing. We loop through each line in the body of the message, scanning for error codes and their attendant strings indicating specificity. The conditional shown in line 5 will match our 451 errors, using the (\w+) pattern to match the string from sendmail indicating what it was trying to do when the fork failed. Once we know what Category this bounce belongs to, we send it to SetType (line 7), which updates the Alert object's state and pre-loads the appropriate Template. Since the message-id of sendmail bounces helpfully include the hostname and timestamp, it's all we need to uniquely identify this Alert. We extract it from the Mail::Header object in line 8, and then pass both of these variables into the Substitution method, which iterates through its list of parameters and assigns them to the _AAA .. _ZZZ patterns in this category's Template, in order. Remember, our template for "sendmail_cannot-fork" Alerts looks like Listing 2c.

There's one conditional for each possible Category which we know about, and a final catch-all line

[2] Like the back of a Volkswagon. Never mind.

```
1 use RedAlert::Client::Sendmail;
2 $alert = new RedAlert::Client::Sendmail;
```

Listing 1b: Code segments from sendmail_client.pl.

```
3 $alert->Configure("/opt/src/redalert/nms_client.conf");
```

Listing 1c: Code segments from sendmail_client.pl.

```
4 use Mail::Header;
5 my $mail = new Mail::Internet( [<>] );
6 my $headers = $mail->head();
7 my $body = $mail->body();
8 $alert->Parse($headers, $body);
```

Listing 1d: Code segments from sendmail_client.pl.

in case we're at the end of our rope and still don't have a match: see Listing 2d.

Back in sendmail_client.pl, we have only to send our completed alert off and we're through.

```
9     $alert->Send;
```

Send creates an eval()-able chunk of perl from the useful parts of our object, namely the Category and Specificity, and fires it off to the server... and that's it! To recap, a minimal RedAlert client, which sends an alert containing the date every time it's invoked, might look like this code from minimal_client.pl:

```
1     use RedAlert::Client;
2     $alert = new Client;
3     $alert->Configure(
            "/path/to/my/client.conf");
4     $alert->SetType("irritating");
5     $alert->Substitution( 'date' );
6     $alert->Send;
```

This assumes that client.conf looks something like this:

```
1     [irritating]
2     template = "minimal_client".
            " was invoked at _AAA"
```

The RedAlert Server, Close Up and Personal

The RedAlert server is the guts of the system. It's the destination for all your different clients' Alerts, and as such needs to know how to track the various categories of alert and when and how to send a notification if a threshold is knocked over. The server (redalert.pl) normally runs as a daemon, but can be put into non-forking mode to facilitate debugging.

Server Configuration File

The RedAlert server config looks syntactically similar to the client configuration file, but contains a lot more information. The section headers are the names of **all** categories, system-wide, for which we might receive an Alert, plus sections for global variables and Panic configs). Unlike the client configuration file, we aren't interested in Templates to fill out for the various Categories; rather, we need to determine what to do when an alert of a particular Category is received. As such, while a client's config may contain only Categories which that particular client will be handling, the server config should contain a section for every category of Alert, system-wide. Again, unconfigured categories indicate that one of your clients is sending mismarked Alerts and will result in a Panic notification. This "last resort" notification means we don't drop anything on the floor, making it easy to track down those elusive "telnet" problems.

All server config files should have a [global] section, to define the port and interface/hostname to bind, an Accounting Interval to summarize and mail statistics reports, and a [panic] section for self-reporting and as an address of last resort for unconfigured categories problems delivering Notifications. Listing 3 shows the start of a typical config (redalert.conf).

```
1 sub Parse {
2     my $self = shift;
3     my ($headerref, $bodyref) = @_;
```

Listing 2a: Code fragments from Sendmail.pm.

```
4     foreach $line (@$bodyref) {
5         if ($line =~ /451: (\w+): cannot fork/) {
6             chop $line;
7             $self->SetType("sendmail_cannot-fork");
8             my $messageid = $headerref->get("Message-ID");
8             $self->Substitution("$1", "$messageid");
9             return $self;
10        }
11        elsif { [ ... ]
12    }
```

Listing 2b: Code fragments from Sendmail.pm.

```
template = "Cannot fork on operation _AAA! Message-ID: _BBB"
```

Listing 2c: Template for sendmail_cannot-fork.

```
12    $self->Panic("Got an unparsable message: @$bodyref");
13    return $self;
14 }
```

Listing 2d: Final catch-all line.

```
[global]
debug                = 1
acct_interval        = 86400s
port                 = 7200
servername           =
          alerthost.ops.domain.com
threshold_interval   = 60
alert_host           =
           mailhost.ops.domain.com

[panic]
alert_type           = email
alert_dest           =
     redalert-admin@ops.domain.com

[sendmail_cannot-fork]
threshold_trigger    = 1
threshold_interval   = 0
alert_dest           =
         duty-pager@ops.domain.com

[sendmail_complicated-threshold]
alert_type           = email
alert_dest           =
          redalert@ops.domain.com
threshold_interval   = 5m
threshold_trigger    = 20
threshold_delta_max  = 5
threshold_delta_min  = 3
threshold_average    = 4
send_only            = 3
send_after           = 2
```

We'll explain what each of the category-specific configuration lines means in a bit; suffice it to say for now that each "451: Cannot Fork" Alert will result in an email page to "duty-pager@ops.domain.com" immediately upon receipt.

Event Loop

redalert.pl uses Joshua Pritikin's excellent Event [4] package to neatly solve some of the thornier problems in running long-term servers in Perl. Event provides "a central facility to watch for various types of events and invoke a callback when these events occur" [from the Event .pod]. A watchable "event" can be a timer-based interval hitting zero, activity on a file or network socket, a signal sent to the process, a variable incrementing or changing its value, to name a few. A callback is simply a subroutine which acts on the information received by its watcher.

The important thing about Event is that its watchers are processed asynchronously, queued for execution and then run sequentially based on their

```
 1 use RedAlert::Server;          # Server methods
 2 use RedAlert::Threshold;       # All the Threshold methods
 3
 4 use Event qw(loop unloop);     # import loop(), unloop() into namespace
 5 require Event::tcpserv;        # JPRIT's server shortcuts
 6 require Event::timer;          # Watcher type for countdowns
 7 require Event::signal;         # Classy signal handlers
 8
 9 $server = new Server;          # instantiate myself
10
11 $configfile = "/opt/src/redalert/rasv.conf";
12
13 # Sanity checking on the values, add a reference to the AppConfig object
14 $server->Configure("$configfile");
15
16 # Create state for each category we'll be monitoring, add a reference
17 # to the Threshold object which contains them.
18 $server->InitThresholds;
19
20 # Daemonize forks us, global_debug from the configfile via AppConfig
21 if ( $server->Daemonize || $server->Config("global_debug") == 1) {
22     my $reread_config = new Event::signal( 'e_signal', 'HUP',
23                          'e_cb', \&ReParse,           # the callback
24                          'e_desc', 'reread_config' );  # description
25    # Server method to update ourselves
26    $server->AddWatcher('reread_config', $reread_config);
27    loop() || $server->Panic("Couldn't start the server's main loop!");
28 }
```

Listing 3: Typical beginning of server, redalert.pl.

priority; this defers the problem of race conditions and is a lot more portable than using threads (which Perl can't do very well anyway).

RedAlert Startup and Watchers

To setup the Event loop, we first import the modules we'll need, then register our starting Watchers (the events we'll be acting upon), and finally start the loop itself. The start of the code for the server is show, in part, in Listing 3 (with commentary inline).

While there are subroutines below for handling different Watchers' callbacks, this section is really the heart of the server program. There's a lot going on here, but it demonstrates a clean object-oriented API. The containment of two other types of objects (RedAlert::Threshold and AppConfig::State) within the RedAlert::Server object lets us encapsulate their internal structure, in accordance with the OO principle of "information hiding." Our Server methods provide a higher-order wrapper to the objects' data, so their structure can change internally (say, to store the Thresholds in a database instead of in memory), but as long as the API remains consistent, the program itself won't notice the difference.

Thresholds

Let's concentrate for the moment on the variables beginning with "threshold_" in the sample config file above. Note that at no time do you specify in the the config what kind of threshold you want; you simply define whatever threshold variables you're interested in watching, and RedAlert does all the work. Let's walk through each of the threshold types in turn, starting with the simplest kind. All threshold variables in the configuration file start with "threshold_"; we'll give a visual representation of each type and then a sample config which would create that kind of threshold.

Summary: The simplest kind of threshold is no threshold at all – if you define an interval but no trigger, RedAlert will simply keep statistics on that Category, sending you a summary of the number of Alerts received over the interval. Summary counters are cleared at the end of the interval. See Figure 1.

In a config file, this is as simple as setting the threshold_trigger to zero, i.e., "don't ever trigger a Notification." If your fall-through threshold_interval in the [global] section is appropriate, zeroing out the trigger is the only thing you need to define. The following lines would create a Summary threshold:

```
[category_type]
threshold_trigger  = 0
```

Trigger: The inverse of the Summary threshold is Trigger, which sends a notification upon each Alert the server receives. This is useful for really desperately bad kinds of problems, which you don't want to track for statistical purposes, you just want to know about it right away. See Figure 2.

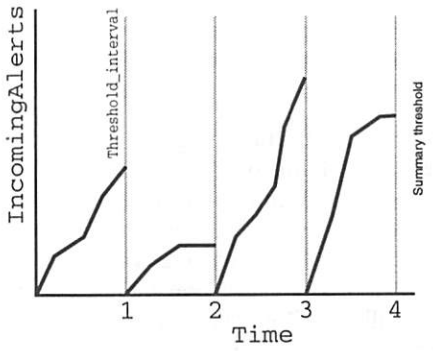


Figure 1: Summary threshold.

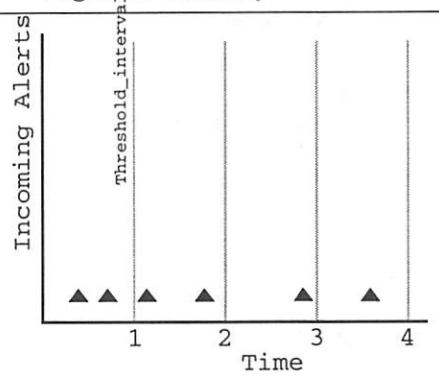


Figure 2: Trigger threshold.

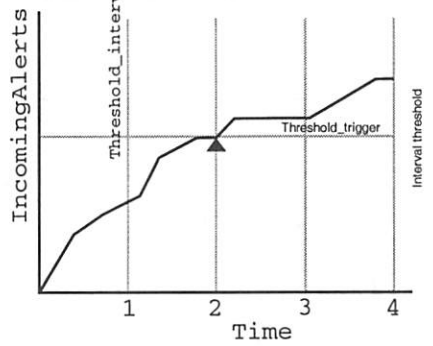


Figure 3: Interval threshold.

To set up this type of threshold, simply set threshold_trigger to be 1.

```
[category_type]
threshold_trigger = 1
```

Interval: An Interval threshold tracks the frequency of incoming alerts and sends a notification if the server receives more than x notifications in y time. A possible use for Interval thresholds is monitoring response times from applications where it's OK for them to sometimes bog down a little bit, but when you receive, say, ten Alerts in ten minutes saying the response time was over five seconds, you'd want to know about it. We'll use these parameters for the graph in Figure 3.

As the graph implies, we'd define both threshold_trigger and threshold_interval for this kind of Alert to be ten, like so:

```
[category_type]
threshold_trigger = 10
threshold_interval = 10
```

Maximum Delta: The greek letter delta signifies change, and with this type of threshold we're monitoring the maximum change in time between receiving two Alerts of the same Category. This is useful for uptime or heartbeat monitoring; you'd run your client as a cron job, say every five minutes, and it would send an "all clear" message once it finished processing. If more than six minutes pass between one notification and the next, we'd want to know about it, which is where Maximum Delta comes in. See Figure 4.

```
[category_type]
threshold_delta_max = 5m
```

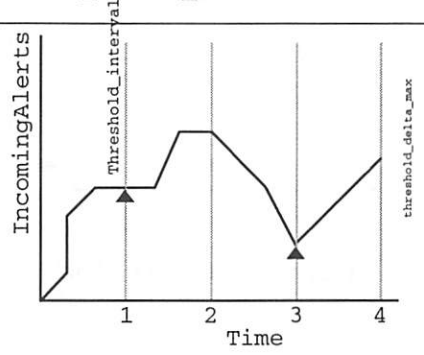


Figure 4: Maximum delta threshold.

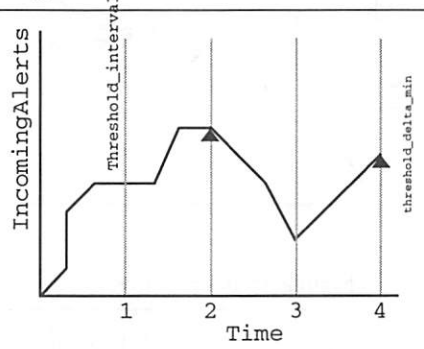


Figure 5: Minimum Delta Threshold

Minimum Delta: Like the Max Delta above, Minimum Delta measures the time between received Alerts of a given Category. However, in this case we want to receive a notification if the Alerts start coming in too quickly instead of too slowly. A potential use for this type of threshold might be monitoring "user not found" errors from a mail server, where a few of them spaced out over a whole day is fairly typical, but ten in rapid succession might indicate a misdirected mailbomb or denial-of-service attack. See Figure 5

```
[category_type]
threshold_delta_min = 3m
```

Average: The most complicated (arithmetically, not configuration-wise!) type of RedAlert threshold keeps a running average of the rate the server is receiving alerts of a particular type, and sends a notification if the average exceeds the configured value. This is quite similar to the idea of "load factor" on UNIX machines, and can be used in the same situations where you'd normally monitor the load average. Average thresholds are more complicated than Interval thresholds because they don't use an absolute value for the trigger; rather, they keep *relative* values and thus provide greater flexibility for monitoring things like proxy transactions per minute.

$$\text{current\_average} = \frac{(\text{interval} - \text{current\_delta}) * \text{old\_average} + 1)}{\text{interval}}$$

The threshold_average config variable is used. It's a numeric value which expresses the number of Alerts received over an interval value. As such, you need to define the threshold_interval as well as threshold_average to configure this type of alert.

```
[category_type]
threshold_interval = 60s
threshold_average = .5
```

Note that thresholds are additive wherever possible: except for ones which trigger a notification on each Alert received or only do statistics-gathering, RedAlert will check the most complex type of threshold first, and then continue down the list to make sure none of the lesser-order thresholds are triggered too. This means that if you define a threshold_trigger and threshold_interval as well as threshold_delta_max for a particular Category, this will create a Maximum Delta threshold plus an Interval threshold for that type of Alert. This allows you to simply and quickly set up a very powerful monitoring profile for your server.

Notifications

So, once a threshold is triggered, what next? There are two kinda of Notifications which can result from a new Alert the server receives: email and SNMP. Each of these methods will prepend its own message onto the text of the Notification, so the worst-case Notification you receive will say something like Listing 4.

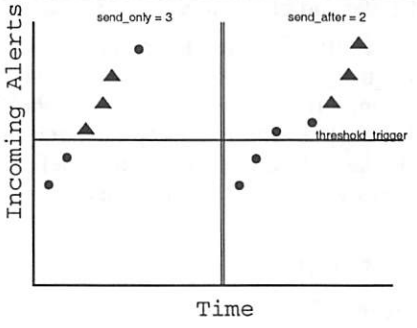


Figure 7: Send After and Send Only

Line 1 is from the Panic method, Line 2 from Server::CheckThresholds, and line three was the original Alert we received.

All categories can specify two attributes which will reduce likelihood of being "spammed" with RedAlert notifications. send_only will increment a counter upon each trap sent, and only send up to this many notifications in one accounting interval. send_after will wait until receiving N alerts *above* a threshold before sending a Notification. See Figure 7. Let's take a closer look at each type of Notification in turn.

Email: Email is pretty self-explanatory; it simply sends an email using the Mail::Internet modules to the SMTP server you specify or to the Panic destination if that gateway is unreachable. To configure email notifications, use lines like these in your server config file:

```
[global]
alert_type = email
alert_dest =
  redalert-admin@ops.domain.com
alert_host =
        mailhost.ops.domain.com

[category_type]
alert_dest =
  category-owner@eng.domain.com
```

The implication here is true: the definitions in the [global] section will act as defaults unless they're overridden by category-specific values. This is known as "translucency" in OO parlance.

SNMP: SNMP Notifications are (by the protocol's very nature) more involved than those of the Email type. They are sent as an SNMP trap to the trap destination port (generally udp/162) of your network management station. The usual object ID of the trap is 1.3.6.1.4.1.43.33.3.9.6, which translates into the "enterprises.a3Com.palm-mib.redalert" branch of the root SNMP-mib2 tree. The last two places are the leaf nodes of the RedAlert MIB: "traps.sendTrap", the value of which trap will look like Listing 5.

If you want to use SNMP traps, you'll first need some kind of network monitoring software like HP OpenView which first, has the ability receive the traps, and second, can be configured to do something useful, like log a message or flash an icon on the screen, upon receipt of said trap. You should compile the included redalert.asn1 MIB definition into your management station's MIB tree and prepare its configuration to receive RedAlert traps (the procedure varies on the software you're using).

In the server's config file, setting "alert_type" to "snmp" requires you to add an "alert_community" variable and transform the meaning of the "alert_dest" line from an email address into an SNMP Object-ID with which to tag the trap. Again, lines in the [global] section will be used as defaults unless you override them in the section for a Category.

```
[global]
alert_type      = snmp
alert_host      =
          traphost.ops.domain.com
alert_dest      =
          1.3.6.1.4.1.43.33.3.9.6
alert_community = public

[category_type]
alert_dest      =
         otherhost.ops.domain.com
alert_community = private
```

RedAlert uses the SNMP::Session [5] module to build up the UDP trap packet, encoding all of these values and adding the Notification message, and then registers it as an Event in the main Event Loop to send as soon as possible. This generates a "queue" of outgoing SNMP alerts, which if not sent within a ten-second timeout, will go into "Panic" mode and send the notification (plus the fact that it couldn't be sent) to your Host of Last Resort.

OTHER: RedAlert is always evolving and maturing, and among the ways it is becoming more useful are additional Notification messages. A Syslog facility is in the works, further refinements might add the ability to dial a modem and speak TAP directly to an alphanumeric paging service. The simple, standard, text-based nature of the Notifications make it very easy to add new backends in response to new technologies or your site-specific requirements.

Gilding the Lily

As a generic Perl program, a RedAlert client can of course undertake whatever additional, non-RedAlert actions you require, such as logging the

```
1 Panic! Couldn't connect to traphost.ops.domain.com sending:
2 Trigger threshold exceeded:
3 Cannot fork on operation SMTP_VRFY! Message-ID:
                <199904291241.FAA17062@mailhost.ops.domain.com>
```

Listing 4: Worst-case notification.

```
Trigger threshold exceeded:
Cannot fork on operation SMTP_VRFY! Message-ID:
      <199904291241.FAA17062@mailhost.ops.domain.com>
```

Listing 5: The traps.sendTrap.

message-ids of "hostname not found" bounce mail. An unusually diligent postmaster whose queues are on a Network Appliance server might save the message-ids and try to pull back the queued messages via snapshot files, but this level of hand-intervention is certainly uncommon in an ISP environment.

An individual running their own site could easily attempt to resubmit bounced mail of certain types using a modified RedAlert client. It would require relatively little effort to write a client which could tally mail bounces for you and compare each bounced recipient and destination against a personal directory of frequent addressees, rewriting and resubmitting such bounces that appear to be simple typos on your part. Postmaster copies of bounces will usually have the body deleted for privacy/security reasons, so this approach works only with the individual recipient copies.

If you are running high-volume services and wish to do sophisticated post-processing of messages, such as rewriting and resubmission, you will probably wish to redirect them to a separate server. Since most filtering methods are non-threaded, and many have clumsy locking, separating the traffic from your important mail server is a reasonable idea.

Extensions for the Future

- Filter messages right in the MTA and call our Sendmail client as a program or local mailer.
- Object Persistence in the form of a database backend to the RedAlert server. This would allow greater reporting and analysis of incoming Alerts, as well as enabling Threshold state to be maintained indefinitely.
- Add SNMP listener to make internal state of client monitoring and configuration available directly.

Availability

The RedAlert code base is released under the GNU Public License, and is available via FTP or by anonymous CVS. If you're interested in using RedAlert, or even in helping to make it more useful by joining the development team, check out the RedAlert homepage at: <URL:http://explosive.net/opensource/redalert/>

The extreme configurability of the client side and the robust, platform independent nature of Perl lead us to believe that the tool will find wide acceptance and use in the systems administration community. Email and Squid monitoring clients exist as of this writing, and we look forward with interest to seeing new clients emerge. For general RedAlert discussion, please send mail to majordomo@explosive.net with "subscribe redalert" in the subject of your message. You will automatically be sent back an authentication token to complete the subscription process, so please make sure your return address is valid.

Acknowledgements

As with any open source project, we stand on the shoulders of giants and add our small contribution to the view. We would like to thank Larry Wall and all the CPAN contributors for their excellent and irreplaceable work in creating the foundation for RedAlert and countless other tools. We also express our gratitude to the pioneers of SNMP, and to 3Com for granting us space in the Enterprise MIB.

Author Information

Eric Sorenson is a UNIX sysadmin and systems programmer currently living the good life in Silicon Valley. When he's not LARTing developers or working on his hardened garage NOC, he enjoys ultralight sailboat racing with his friends and fiancee and listening to experimental ambient music. Reach him electronically at <eric@explosive.net>.

Strata Rose Chalup <strata@virtual.net> has been a sysadmin professionally since 1983, long before all the Internet hype. For the past several years, she has specialized in high-volume network service architecture, particularly for email. Following a road paved with good intentions, Strata has drifted into the shady netherworld of Project Management and looks forward to resuming more hands-on technical work. All year she has been promising herself "just one more contract" before taking a month off to catch up on scuba diving, learning Perl, and getting back into tai chi. In her minimal spare time, Strata is learning to program her new digital camera and working with her husband to convert a 24-foot school bus into a custom motorome for dive weekends up and down the California coast.

Inline References

[1] Data::Dumper 2.101 by Gurusamy Sarathy, <gsar@umich.edu>.

[2] MailTools 1.13 by Graham Barr, <gbarr@pobox.com>.

[3] AppConfig 1.52 by Andy Whardley, <abw@cre.canon.co.uk>.

[4] Event 0.51 by Joshua Pritikin, <bitset@mindspring.com>.

[5] SNMP_Session 0.70 by Simon Leinen <simon@switch.ch>.

Other References

Chalup, Hogan, et al., "Drinking From the Fire Hose," 1998 LISA.

Pree, Wolfgang, "Design Patterns for Object-Oriented Software Development," 1995, ACM Press.

Friedl, Jeffrey E. F. "Mastering Regular Expressions," 1997, O'Reilley and Associates.

Srinivasam, Srinam. *Advanced Perl Programming*, 1997, O'Reilley and Associates.

Comprehensive Perl Archive Network (CPAN),
AppConfig-1.52 by Andy Whardley <abw@cre.canon.co.uk>,
Data::Dumper-2.101 by Gurusamy Sarathy <gsra@umich.edu>,
Event-0.51 by Joshua Pritikin <joshua.pritikin@db.com>,
Event-tcp-0.007 by Joshua Pritikin <joshua.pritikin@db.com>,
MailTools-1.53 by Graham Barr <gbarr@pobox.com>,
SNMP_Session-0.70 by Simon Leinen <simon@switch.ch>.

Deconstructing User Requests and the Nine Step Model

Thomas A. Limoncelli – Lucent Technologies/Bell Labs

ABSTRACT

How can we improve the process by which System Administrators (SAs) help users? SAs spend much of their time responding to requests from users. Better system administrators use a similar, structured, process. I present the structured process as I have seen and practiced it, examples of each step in the process, and the pitfalls of eliminating various steps. Finally I look at the paper in the larger context of a step towards improving the science of System Administration.

Introduction

In this paper I document and analyze the process for resolving trouble reports that is used by the best system administrators (SAs) I have known and observed. The goal is to improve the process by which SAs repair problems that are reported by users (e.g., "helpdesk requests"). This paper also establishes a base model for use in future studies.

A large part of a SAs workload comes from users that report problems, request improvement, ask questions, and so on. This paper is focused on resolving these requests as efficiently as possible given the resources available. Thus, retain user happiness. The model will identify requests that are out of scope (request for new features, questions, etc.) and offer appropriate responses.

The method to process these "trouble reports" has nine steps, which can be grouped into four phases:

- Phase A: The Greeting ("Hello")
  - Step 1: The Greeting
- Phase B: Problem Identification ("What's wrong?")
  - Step 2: Problem Classification
  - Step 3: Problem Statement
  - Step 4: Problem Verification
- Phase C: Planning and Execution ("Fix it")
  - Step 5: Solution Proposals
  - Step 6: Solution Selection
  - Step 7: Execution
- Phase D: Verification ("Verify it")
  - Step 8: Craft verification
  - Step 9: User Verification/Closing.

In addition to being useful to SAs, I have found that if users understand this model they assist the process. They become more skilled in getting the help they desire. They will be prepared with the right information and they can nudge the SA along through the process if necessary.

I should point out that I don't feel this process is a panacea, nor do I think this process is a replacement for a creative mind or technical experience. However, this gives SAs a common set of terminology and a well tested, effective process, to use when interacting with users. SAs will not magically all become equally productive. Creativity, experience, resources, tools, personal and external management are other influences that contribute to productivity.

Historical Comparison

At the close of World War II, the United States found itself with a huge excess of manufacturing capacity. As a result, companies started producing hundreds of new products that households and businesses never had access to previously. The thousands of returning G.I.'s found jobs selling these new products. The new manufacturing capacity, the new products, and the large number of returning G.I.'s looking for work combined to produce a new era for the U.S. economy. In short, the large manufacturing capacity met the large demand which met the large sales force.

As time went on competition grew. Companies found that it was no longer sufficient to have a large sales force, a good sales force was needed. They started to ask, "What makes the high performing salesmen different from the others?" At the request of industry, business schools began studying salespeople.

Industry encouraged business schools to increase their study of the sales process. They discovered that the better salesmen, whether or not they realized it, had a specific, structured method they employed. It involved specific phases or steps. Mediocre salespeople deviated from these phases in varying ways or performed certain phases badly. The low performers had little or no consistency in their methods.

The method, once identified, could be taught. Thus sales skills went from an intuitive function to a formal function with well-defined parts. Previous sales training mostly consisted of explaining the product's features and qualities. Subsequently, training included exploration of the selling process.

This deconstruction permitted further examination and therefore further improvement. Each step

could be studied, measured, taught, practiced, and so on. Focus is improved because a single step can be studied in isolation. Also the entire flow of steps could be studied (a holistic approach).

I imagine that if anyone explained the structured process to the high performing salespeople it would sound strange. To them it comes naturally. It would be like explaining to Picasso how to paint. However, to the beginner, this framework gives structure to a process they are learning.

In recent years, system administration has begun a similar journey. SA previously was a craft or an art practiced by few people. With the recent, explosive growth of corporate computing and intranet/internet applications, the demand for SAs has been similarly explosive. A flood of new system administrators has arrived to meet the need. Quality of their work varies. Training often takes the form of teaching particular product features, similar to when a salesperson's training consisted mostly of learning the product line. Other training methods include exploring manuals and documentation, trial by fire, training by social institutions (IRC, mailing lists, etc.) and professional institutions (SAGE, SANS, etc.).

SA also needs to have its processes understood. Recently there has been a rise in in-school curricula being taught on the topic of system administration. However, most of it has been specific to particular technologies and vendors rather than being non-vendor-specific and theoretical. I hope that more theoretical models will be introduced and popularized in the coming years and this will result in large improvements similar to the improvements made in the sales profession.

The Process

The process described in this paper contains nine steps grouped into four phases. As seen in Figure 1, the phases deal with:

A. how the user reports the problem,
B. identifying the problem,
C. planning and executing a solution, and
D. verifying that the problem resolution is complete.

Readers should be forewarned that sometimes certain steps are iterated as required. For example, during Step 4 (Problem Verification) the SA may realize the issue has been misclassified and one must return to Step 2 (Problem Classification). This can happen at any step and require returning to any previous step.

A description of the steps follows.

Phase A: The Greeting ("Hello!")

The first phase only has one deceptively simple step. The user reports the problem.

Figure 2: Greeting phase.

Step 1: The Greeting – "The Greeter"

This step is where someone or some thing asks, "How may I help you?" This step includes everything related to how the user's request is submitted. Commonly users can report a problem by calling a customer care center, by walk-up "help desk," or electronic submission. These methods are called "Greeters." Multiple greeters are needed for easy and reliable access. If the user's problem is that they can't send email, reporting this via email is not possible. Having multiple greeters is valuable.

Sometimes problems are reported by automated means rather than by humans. For example, network monitoring tools such as "mon" [Trocki], HP OpenView, and Tivoli can notify SAs that a problem is occurring. This is still the same process, although some of the steps may be expedited by the tool.

Every site and every user is different. Greetings become more or less appropriate based on many factors. Is the user local or remote? Is the user experienced or new? Is the technology complicated or simple? These questions can help a site select which greeters should be used.

How do users know how to find help? There are various ways to advertise the available greeters. Examples include: signs in hallways, newsletters,

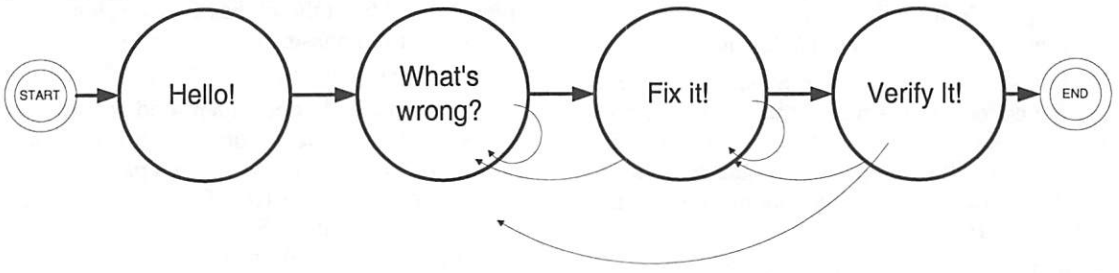


Figure 1: General flow of problem solving.

stickers on computers or phones, and even banner advertisements on internal web pages.

Summary of common methods used to report problems (certainly this is an incomplete list):

- Phone
- Email
- Walk-up helpdesk
- Visiting SAs office
- Submission via web
- Submission via custom application
- Report by automated monitoring system

Phase B: Problem Identification ("What's wrong?")

The second phase is focused on classifying the problem, recording it, and verifying the problem.

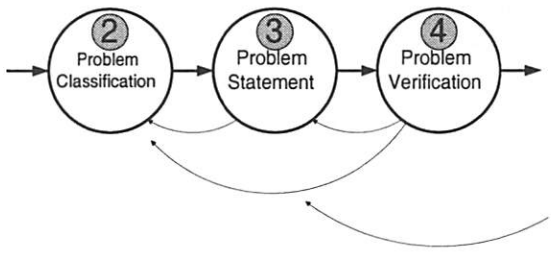


Figure 3: What's wrong?

Step 2: Problem Classification – "The Classifier"

In this phase, the request is classified to determine who should handle the request. This role is called "The Classifier." The classifier may be a human or automated. For example, at a walk-up help desk, staff might listen to the problem description to determine its classification. A phone response system may ask the user to press 1 for PC problems, 2 for network problems, etc. If certain customer groups are helped by certain SAs, their requests may be automatically forwarded based on the requester's email address or employee id number.

When the process is manual, a human must have the responsibility to surmise the classification from the description or to ask the user more questions. A formal decision tree may be used to determine the right classification.

No matter how the classification is done, the user should be told how the request is classified. This creates a feedback loop that can detect mistakes. For example, if a classifier tells a customer, "This sounds like a printing problem. I'm assigning this issue to someone from our Printer Support Group." the user retains greater participation in the process. The user may point out that their problem is more pervasive than just printing, leading to a classification such as a network problem.

If a phone response system is used, the user has classified the request already. However, they may not be the best person to make this decision. The next human that speaks with the user should be prepared to validate the user's choice in a way that is not insulting. Also, the choices given to the user must be carefully constructed and revised over time.

Many requests may be transferred or eliminated at this stage. For example, if the user is requesting a new feature they should be transferred to the appropriate group that handles requests for features. That role is often called "service management." Or if the request is outside the domain of work that the support structure does, they might be referred to another department. Or if the request is against policy and therefore it must be denied. The issue may be escalated to management if the user disagrees with the decision. For this reason, it is important to have a well-defined scope of service and process for requesting new services.

Step 3: Problem Statement – "The Recorder"

This is where the user states the problem with full details and this information is recorded. This person performing this role is called "The Recorder" and is often the same person as the classifier. The skill required by the recorder in this phase is the ability to listen and ask the right questions to draw out the needed information from the user. The recorder extracts the problem statement and records it.

A problem statement describes the problem being reported and enough clues to reproduce and fix the problem. A bad problem statement is vague or incomplete. A good problem statement is complete and identifies all hardware and software involved as well as their location, the last time it worked, and so on. Some times not all of that information is appropriate or available.

An example good problem statement is "PC talpc.example.com (a PC running Windows NT 4 SP4) located in room 301 can not print from MS-Word97 to printer 'rainbow', the color PostScript printer which is located in room 314. It worked fine yesterday. It can print to other printers. The user does not know if other computers are having this problem."

It is unreasonable to expect problem statements directly from users to be so complete. They require assistance. The above problem statement comes from a real example where a customer sent email to a SA that simply stated, "Help! I can't print." That is about as ambiguous and incomplete as a request can be. A reply was sent asking, "To which printer? Which PC? What application?"

The reply included a statement of frustration. "I need to print these slides by 3 pm, I'm flying to a conference!" At that point, email was abandoned and the telephone was used. This permitted a faster "back and forth" between the user and classifier. No matter the medium, it is important that this dialog take place and that the final result be reported to the customer.

Sometimes the recorder can perform a fast loop through the next couple steps to accelerate the process. The recorder might ask the typical "just in case"

questions such as "Is it plugged in?" and "Have you check the manual or on-line help?" or "Did you receive the memo that said printer 'rainbow' would be decommissioned last week?" In our example the user indicated that there was an urgent need to have the slides printed. Here it might be appropriate to suggest using a different printer that is known to be working.

Certain classes of problems can be completely stated in a simple way. I have found that internet routing problems can best be reported by listing two IP addresses that can not ping each other, but which can both communicate to other hosts and including a traceroute from both (if possible) host to the other.

Large sites often have different people recording requests and executing the requests. This added "hand-off" introduces a challenge as the recorder may not have the direct experience required to know exactly what to record. In that case, it is prudent to have pre-planned sets of data to gather for various situations. For example, if the user is reporting a network problem, the problem statement must include an IP address, the room number of the machine that is not working, etc. If the problem relates to printing one might be required to record the name of the printer, the computer generating the print job, the application generating the print job, etc.

Most sites use some kind of "trouble ticket" software to record the user's report. It can be useful if the software requests different information depending on how the problem has been classified.

Step 4: Problem Verification – "The Reproducer"

This is where the SA tries to reproduce the problem. This role is called "The Reproducer." If the problem can not be reproduced, often the problem being reported is not being properly communicated and one must return to Step 3 (Problem Statement). If the problem is intermittent, then this process becomes more complicated but not impossible.

It is critical that the method used to reproduce the problem is recorded for later repetition in Step 8 (Craft Verification). Encapsulating the test in a script will make verification easier. One of the benefits of command-driven systems like UNIX is the ease in which such a sequence of steps can be automated. Graphical user interfaces make this phase more difficult since there is no way to automate or encapsulate the test.

The scope of the verification procedure must not be too narrowly focused nor too wide, nor mis-aimed. If the tests are too narrow, the entire problem may not be fixed. If the tests are too wide, the SA may waste time chasing non-issues.

It is possible that the focus may be mis-aimed. There may be another, unrelated problem in the environment that is discovered while trying to repeat the user's reported problem. Some problems can exist in an environment without being reported or without affecting users. It can be frustrating for both the SA and the user if many unrelated problems are discovered and fixed along the way to resolving an issue. If an unrelated problem is discovered that is not in the critical path, it should be recorded so that it can be fixed in the future. On the other hand, determining if it is in the critical path is difficult, so fixing it may be valuable. Alternatively, it may be a distraction or may change the system enough to make debugging difficult.

Sometimes direct verification is not possible or even required. If a user reports that a printer is broken the verifier may not have to reproduce the problem by attempting to print something herself. It may be good enough to verify that new print jobs are queuing and not being printed. Such superficial verification is fine in that situation.

However, other times exact duplication is required. The verifier might fail to reproduce the problem on her own desktop PC, and may need to duplicate the problem on the user's PC. Once the problem is duplicated in the user's environment, it can be useful to try to duplicate it elsewhere to determine if the problem is local or global. A lab of equipment for the purpose of reproducing reported problems may make supporting remote users or complicated products easier.

Phase C: Planning and Execution ("Fix it")

In the previous phase the problem was identified. In this phase it is fixed. This involves planning possible solutions, selecting one, and executing it.

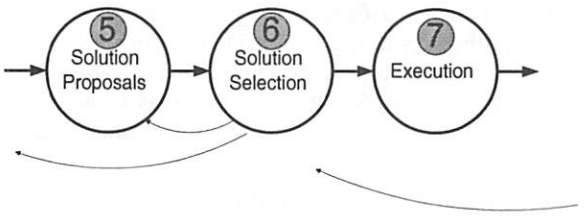


Figure 4: Flow of repair.

Step 5: Solution Proposals – "Subject Matter Expert"

This is the point where the possible solutions are enumerated. This role is performed by the "Subject Matter Expert" or SME. Depending on the problem, this list may be large or small. For some problems the solution may be obvious and there is only a single proposed solution. Other times there are many possible solutions. Often verifying the problem in the previous step helps finding possible solutions.

The "best" solution varies depending on context. At a bank, the Help Desk's solution to a client-side NFS problem was to reboot. It was faster than trying to fix it and it got the customer up and running quickly. However, in a research environment, it would make sense to try to find the source of the problem, perhaps unmounting and re-mounting the NFS mount that reported the problem. In our printing example,

since the user indicated that they needed to leave for the airport soon, it might be appropriate to suggest alternative solutions such as recommending a different printer that is known to be working. If the user is an executive flying from New Jersey to Japan with a stop-over in San Jose, it might be reasonable to transfer the file to an office in San Jose where it can be printed. A clerk could hand the printout to the executive while he waits for his connecting flight at the San Jose airport.[1]

Some solutions are more expensive than others. Any solution that requires a desk-side visit is generally going to be more expensive that one that can be handled without such a visit. This kind of feedback can be useful in making purchasing decisions. Lack of remote support capability affects the total cost of ownership of a product. There are tools (commercial and non-commercial) that add remote support to such products.

If the SA does not know any possible solutions the issue is escalated to other, usually more experienced, SAs.

Step 6: Solution Selection

Once the possible solutions are enumerated, one of them is selected to be attempted first (or next, if we are looping through these steps). This role is also performed by the "Subject Matter Expert" or SME.

Selecting the best solution tends to be either extremely easy or extremely difficult. However, solutions often can not be done simultaneously so possible solutions must be prioritized, usually with the help of the user.

The user should be included in this decision. The user has a better understanding of their own time pressures. If the user is a commodities trader, she or he will be much more sensitive to downtime during the trading day than, say, a technical writer or even a developer (provided they're not on deadline). If solution A fixes the problem forever but requires downtime, and solution B is a short-term fix, the user has to be consulted as to whether A or B is "right" for his or her situation. It is the responsibility of the SME to explain the possibilities. Some of this the SA should know based on his or her environment. There may be predetermined service goals for downtime during the day. SAs on Wall Street know that downtime during the day can cost millions, so sort-term fixes may be selected and a long-term solution may be scheduled for the next maintenance window. In a research environment, the rules about downtime are more relaxed and the long-term solution may be selected immediately.[2]

When dealing with more experienced users, it can be useful to let them participate in this phase. They may have useful feedback. In the case of inexperienced users, it can be intimidating or confusing to hear all these details. It may even unnecessarily scare them. For example, listing every possibility from a simple configuration error to a dead hard disk may cause the user to panic and is a generally bad idea. (Especially when the problem turns out to be a typo in CONFIG.SYS)

Even though the user may be inexperienced they should be encouraged to participate in determining and choosing the solution. This can help educate the user so future problem reports can flow more smoothly or even help them solve their own problems in the future. It can also give the user a sense of ownership, the warm fuzzy feeling of being part of the team/company, not a "user." That can help break down the "us vs. them" mentality that is common in industry today.

Step 7: Execution – "The Craft Worker"

This is where the solution is attempted. The skill, accuracy, and speed at which this step is completed is dependent on the skill and experience of the person executing the solution.

The term "craft worker" refers to the SA, operator, or laborer that performs the technical tasks involved. This term comes from other industries. For example, the foreman at a construction site plans what is done when, the craft workers (carpenters, plumbers, etc.) do the physical work. In the telecommunications industry, while others have received the order and planned the provisioning of the service, the craft workers run the cables, connect circuits, etc. In a computer network environment, the Network Architect might be responsible for planning the products and procedures used to give service to customers, but when a new ethernet interface needs to be added to a router, the craft worker installs the card and configures it.

Even the user might become the craft worker. This is particularly common when the user is remote and is using a system with little or no remote control. In that case, the success or failure of this step is in the hands of this user.

A dialog is required between the SA and the user to make the solution work. Has the user executed the solution properly? If not, are they causing more harm than good?

The dialog has to be adjusted based on the skill of the user. It can be insulting to spell out each command, space, and special character to an expert user. It can be intimidating to a novice user if the SA rattles off a complex sequence of commands. Asking, "What did it say when you typed that?" is better than "Did it work?" in these situations. Bi-directional communication is critical and the skills related to this can be a unique specialty in our industry. Training is available. Workshops that focus on this area often have titles that

[1]This is a true story which happened at a previous employer. The printer was a very expensive plotter. Only one such plotter was at each company location.

[2]Personal communication with Josh Simon.

include the buzzwords "Active Listening," "Interpersonal Communication," "Interpersonal Effectiveness," or simply "Advanced Communication."

At this point it is tempting to think that we are done. However, we aren't done until the work has been checked and the user is satisfied. That brings us to the final phase.

Phase D: Verification ("Verify it")

At this point the problem should be remedied but we need to verify that it really has been. This phase isn't done until the customer agrees the problem is fixed.

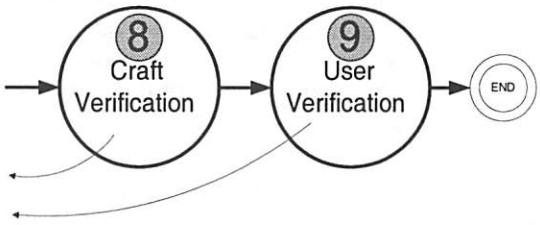


Figure 5: Verification flow.

Step 8: Craft Verification

This is the step where the craft worker that executed Step 7 (Execution) verifies that the actions taken to fix the problem were successful. If the process used to reproduce the problem in Step 4 (Problem Verification) is not recorded properly, or not repeated exactly, the verification will not properly happen. There is potential that the problem still exists, but verification fails to demonstrate this, or the problem may have gone away but the SA does not know this.

If the problem still exists, return to Step 5 (Solution Proposals) or possibly an earlier step.

One tool that is useful in this step is "diff." Capture the output generated when the problem is reproduced. During craft verification first one may "diff" the captured output against the new output. Alternatively, one might copy the output that demonstrates the problem and edit it to be the way it should be on a working system. Or one might have a working system to generate a sample "good" output. Either way, "diff" can then be used to compare the current output with the corrected output.

Variations on this theme are many. Once a user was able to provide me with a sample TeX file that processed fine in his previous department's TeX installation but not on ours. Since I had an account on the computers of his previous department, I could establish a basis for comparison. This was extremely useful. Eventually I was able to fix our TeX installation through successive refinement of the problem and comparison on both systems.

Some problems do not generate output that is well suited to "diff," but perl and other tools can pare down the output to make it more palatable to diff. Once we were tracking reports of high latency on an ISDN link. The problem happened only occasionally. We set up a continuous (once per second) "ping" between two machines that should demonstrate the problem. We recorded this output for a number of hours and observed consistently good (low) latency except occasionally there seemed to be trouble. We built a filter in awk that would extract pings with high latency (where latency was more than three times the average of the first 20 pings) and would reveal missed pings. We noticed that no pings were being missed, but every so often a series of pings took much longer to arrive. We used a spreadsheet to graph the latency over time. Visualizing the results helped us notice that the problem occurred every five minutes within a second or two. It also happened at other times, but every five minutes we were assured of seeing the problem. We realized that there are protocols that do certain operations every five minutes. Could a route table refresh be overloading the CPU of a router? Maybe there was a protocol that overloaded a link? By repeating the ping test between smaller and smaller portions of the path, we were able to isolate which router was introducing the latency. Its CPU was being overloaded by routing table calculations, which happened every time there was a real change to the network or every five minutes. This agreed with our previously collected data. The fact that it was an overloaded CPU, not an overloaded network link explains why latency increased but no packets were lost. Once we fixed the problem with the one router we used our ping test and filter to demonstrate that the problem had been fixed.

Step 9: User Verification/Closing – "The User"

The final step is for the user to verify that the issue has been resolved. If they aren't satisfied, the job isn't done. This role is performed by the user themselves.

Presumably if the craft worker positively verified that the solution worked (Step 8, Craft Verification) this should not be needed. However, often users report at this point that the problem still exists. This is such a critical problem at some sites that the author chose to emphasize it by making it a separate step.

User verification can reveal mistakes made in previous phases. Communication problems include the user not properly expressing the problem, the SA not understanding the user, or the SA not properly recording the problem. Errors may have crept into the planning phase. The problem that was verified in Step 4 (Problem Verification) may have been a different problem that also exists or the method that verified the problem may have been incomplete. The solution may not have fixed the entire problem or may have turned the problem into an intermittent one.

In either case, if the user does not feel the problem is fixed, there are many possible actions. Obviously, Step 4 (Problem verification) should be repeated to find a more accurate method to reproduce

the problem. However, at this point it may be appropriate to return to other steps. For example, the problem could be re-classified (Step 2, Problem Classification) or re-stated (Step 3, Problem Statement), or escalated to more experienced SAs (Step 5, Solution Proposals). If all else fails, one may have to resort to escalating the problem to management.

It is important to note that "verification" isn't to verify that the user is happy, but that the user's request is satisfied. Some users are never happy. In a perfect world, this step would be where the customer thanks the SA, but we know we can not always expect gratitude. Sometimes gratitude takes odd forms because users may not understand what is "hard" and what is "easy." The typical example is the user that hardly blinks when SAs work overtime to resolve a major network issue but send compliments to the SA's management after being so impressed that the SA fixed a problem where the user couldn't log in (the caps lock key had been pressed).

Once user verification is complete, the issue is "closed." If a tracking system is used, the "ticket is closed." Lastly, and possibly only in a perfect world, the customer is told to have a nice day.

Perils of Skipping A Step

Each step is important. If any step in this process is performed badly the process can break down. It is my experience that many SAs skip a single step either due to lack of training or honest mistake. I find many stereotypes about bad SAs are the results of SAs that skip a particular step. I have assigned Seinfeld-esque names to each of them and list possible ways of improving the SAs process. Reading this paper should also help improve their process.

The Ogre: Grumpy, caustic SAs are trying to scare users from Step 1. They are preventing the Greeting from happening. Suggestion: Management must set expectations for friendliness. Also, it is important to set expectations with users.

The Mis-delegator: If you've called a large company's technical support line and the person that answered the phone refused to direct your call to the proper department, you know what its like to deal with a Mis-delegator. They skip Step 2. Suggestion: A formal decision tree of what issues are delegated where.

The Assumer: I've never seen anyone habitually skip Step 3, but I've seen SAs assume they understand what the problem is when they really don't. Suggestion: An "Active Listening" class usually helps this kind of SA.

The Non-Verifier: A SA that skips problem verification (Step 4) is usually off fixing the wrong problem. Recently I was panicked by the news that "the network was down." In reality, a non-technical user couldn't read their email and reported "the network is down." This claim hadn't been verified by the newly hired SA who hadn't yet learned that certain novice users report all problems as "the network is down." The user's email client was misconfigured. Suggestion: Teach SA to replicate problems, especially before escalating them.

The Wrong Fixer: Inexperienced SAs sometimes are not creative, or are too creative, in proposing and selecting solutions (Step 5 and 6). But skipping these steps entirely results is a different issue. After being taught how to use an Ethernet monitor (a network sniffer), an inexperienced but enthusiastic SA was found dragging out the sniffer no matter what problem was being reported. He was a Wrong Fixer. Suggestion: Mentoring or training. Increase the breadth of solutions with which the SA is familiar.

The De-Executioner: Incompetent SAs sometimes cause more harm than good when they execute incorrectly. How embarrassing to apply a fix to the wrong machine. However, it happens. Suggestion: Train the SA to check what they have typed before pressing RETURN or clicking "OK." It can be useful to include the hostname in one's shell prompt.

The Hit-And-Run Sysadmin: This SA walks into a user's office, types a couple keystrokes and waves as he walks out the door saying, "That should fix it." The users are frustrated to discover that the problem was not fixed. In all fairness, what was typed *really should have fixed the problem* but it didn't. Suggestion: Management needs to set expectations on verification.

The Closer: Some SAs are obsessed with "closing the ticket." Often SAs are judged on how quickly they close tickets. In that case, they are pressured to skip the final step. I borrow this name from the term used to describe high-pressure salespeople who are focused on "closing the deal." Suggestion: Management should not measure performance based on how quickly issues are resolved but on a mixture of metrics that drive the preferred behavior. Metrics should not include time waiting for customers when calculating how long it took to complete the request. Tracking systems permit a request to be put into a "customer wait" state while waiting for them to complete actions, etc.

Improving The Process

With the process broken into specific steps, each grouped into distinct phases, improvements can be made by focusing on each step. Entire books could be written on each step. This has happened in other professions that have similar models (Nursing, Sales, etc.).

In addition to focusing on improving each step, one may also focus on improving the entire process. Transitioning to each new step should be fluid. If the user sees a staccato hand-off between each step, the process can look amateurish or disjointed.

Every hand-off is an opportunity for mistakes and miscommunication. The fewer hand-offs, the fewer opportunities for mistakes. A site small enough to have a single SA has zero opportunities for this class of error. However, as systems and networks grow and become more complicated, it becomes impossible for a single person to understand, maintain, and run the entire network. As a system grows hand-offs become a necessarily evil. This explains a common perception that users have: larger SA groups are not as good as smaller ones. However it shows an area for improvement: when growing a SA group one should focus on maintaining high quality hand-offs. Or, one might choose to develop a "single point of contact" (SPOC) or user advocate for an issue. That results in the users seeing a single face for the duration of a problem.

In addition to improving the individual steps or the flow, one can take a holistic view to seek improvements. No man is an island, and no single trouble report is an island either. The flow from problem report to problem report is an area that should be studied. Does a user report the same issue over and over? (Why is it recurring?) Always in a particular category? (Is that system badly designed?) Are many users reporting on the same issue? (Can they all be notified at once? Can that problem receive additional priority?) All of these scenarios can be identified and become areas of improvement for a SA organization.

For example, during a major network outage, many users may be trying to report problems. If users report problems though a automatic phone response system ("Press 1 for... press 2 for...") usually such a system can be programmed to announce the network outage before listing the options. "Please note the network connection to Denver is currently experiencing trouble. Our service provider expects it to be fixed by 3 pm. Press 1 for... press 2 for...." This kind of "global announcement" can be easily provided in any of the first three steps.

If the users talk to a different person every time they call for support, there is less chance for the SA to become familiar with the users' particular needs. There are ways of rectifying this. For example, subteams of the SA staff may be designated to particular groups of users, rather than based on which technology they support. If the staff answering the phone is extremely large they may be using a phone "Call Center" system where users call a single number and the call center routes the call to an available operator. Modern call center systems can route calls based on caller id. They can use this functionality to, for example, route the call to the same operator they spoke to last time if that person is available. This means there will be a tendency for users to be speaking to the same person. It can be comforting to be speaking to someone that recognizes your voice.

A better educated user can be a better customer. If a user understands the nine steps that will be followed, they can be better prepared when reporting the problem. They might have more complete information about the problem being reported when they call because they understand the importance of complete information. In gathering this information, they will have narrowed the focus of the problem report. They might have specific suggestions on how to reproduce the problem. They may have narrowed the problem down to a specific machine or situation. Their additional preparation may lead them to solve the problem on their own! Training for users should include explaining the nine step process to facilitate interaction between users and SAs.

Some things hurt the process. For example, an ill-defined delineation of responsibilities makes it difficult for a "classifier" to delegate the issue to the right person. Inexperienced "recorders" don't gather the right information in Step 3 (Problem Statement) which makes further steps difficult and may require contacting the user unnecessarily. A list of standard information to be collected for each classification will reduce this problem.

Architectural decisions may impede the classification process. The more complicated a system is, the more difficult it can be to identify and duplicate the problem. Sadly, some well accepted software design concepts are at odds with this, such as delineating a system into layers. For example, a printing problem in a large UNIX network could be a problem with DNS, lpd on the servers, lpr on the client, the wrong version of lpr, misconfigured user environment, the network, BOOTP, the printer's configuration or occasionally even the printing hardware itself. Typically many of those layers are maintained by separate groups of people. To diagnose the problem accurately requires the SAs to be experts in all of those technologies, or that the layers crosscheck each other.

Team of One

The solo SA can still benefit from using the model to make sure that users have a well-defined way to report problems, that problems are recorded and verified, solutions are proposed, selected and executed, and that both the SA and the user has verified the problem is resolved.

Problems can be escalated to vendor support lines. Often the solo SA's site is part of a larger company that has a larger IT organization.

Future Work

I feel that deconstructing and analyzing the things that SAs do is the most fruitful way to improve our profession and turn our practice into a science. I hope to see other processes analyzed this way. I also look forward to competing models that describe what I have presented here. Such an academic debate would only help our profession. I would also like to see extensions to the model or exploration of ways to perform particular stages.

The System Administration Maturity Model (SAMM) presented in [Kubi93] establishes a maturity model for IT that is similar to CMU's Software Maturity Model. It would be fruitful to explore how SAMM and the process described in this paper can complement each other.

This paper does not discuss metrics. A system of metrics grounded in this model might detect areas needing improvement. The model can be instrumented easily to collect metrics. However, developing metrics that drive the right behaviors is difficult. For example, if SAs are rated by how quickly they close tickets, one might encourage "The Closer" behavior described above.

As SAs pro-actively prevent problems, reported problems will become more serious and time consuming. If average time to completion grows, does that mean minor problems were eliminated or that SAs are being slower at fixing all problems?

Many other questions need further research:

- What are all the ways to greet users? How do they compare by cost, by speed (faster completion), by user preference? Is the most expensive method the one that users prefer the most?
- How can classification be improved?
- Some problem statements can be stated concisely, like the routing problem example in Step 3. Given various situations, what is the shortest problem statement that completely describes the issue?
- Are there times not to use these steps? For example, if a router has lost power, there is no need to go through the steps. One simply turns the power back on!
- Diagnostic tools that integrate well with this model.
- What is the best way to communicate status to a single user? To many users?
- Which tools are good matches to this model? What tools are missing?
- Some SAs feel that after a problem is fixed, one should reboot the host and verify that the problem doesn't reappear. Other operating systems are known to have most common problems fixed via a reboot. How do these situations fit into the model?

Conclusion

I have presented a model that deconstructs the process of users requesting and receiving support in hopes of making the process repeatable, easier to teach, and easier to improve and manage. The process has four phases: "The Greeting," "Problem Identification," "Planning and Execution," "Fix and verify." Each phase has distinct steps.

By following this model the process becomes more structured and formalized. The process is something highly akin to the scientific process: observe, hypothesize, test, repeat.

Phase	Steps	Role
Phase A "Hello!"	Step 1: The Greeting	Greeter
Phase B "What's wrong?"	Step 2: Problem Classification	Classifier
	Step 3: Problem Statement	Recorder
	Step 4: Problem Verification	Reproducer
Phase C "Fix it"	Step 5: Solution Proposals	Subject Matter
	Step 6: Solution Selection	Expert
	Step 7: Execution	Craft
Phase D "Verify it"	Step 8: Craft Verification	Craft
	Step 9: User Verification / Closing	Customer

Figure 6: Overview of problem solution phases.

Analyzing the execution of each step as well as viewing the entire process holistically are fruitful sources for improving the way user requests are handled in an organization. In addition, having a process makes measurement possible.

The nine steps should be integrated into training programs for SAs. If all SAs used the same terminology to describe their processes it would help communication between SAs. While knowledge of the model can improve a SA's effectiveness by leveling the playing field, it is not a panacea; nor it is a replacement for a creativity, experience, having the right resources, etc. Users that understand these steps can be our best customers because they become part of the process.

Deconstructing the process has permitted a deeper analysis of this important portion of our field. Other parts of system administration could benefit from similar analysis.

Acknowledgements

I would like to thank Eric Anderson, Josh Simon, Tommy Reingold and Jay Stiles for their editing, feedback and suggestions.

References

[Arch93] Archer, Barrie, "Towards a POSIX Standard for Software Administration," Systems Administration (LISA VII) Conference, Monterey, CA, pp. 67-79, 1993.

[Bent93] Bent, Wilson, "System Administration as a User Interface: An Extended Metaphor," Systems Administration (LISA VII) Conference, Monterey, CA, pp. 209-212, 1993.

[Hunt93] Hunter, Tim and Watanabe, Scott, "Guerrilla System Administration," Systems Administration (LISA VII) Conference, Monterey, CA, pp. 99-105, 1993.

[Kubi92] Kubicki, Kubicki, "Customer Satisfaction Metrics and Measurement," Systems Administration (LISA VI) Conference, Long Beach, CA, pp. 63-68, 1992.

[Kubi93] Kubicki, Carol, "The System Administration Maturity Model – SAMM," Systems Administration (LISA VII) Conference, Monterey, CA, pp. 213-225, 1993.

[Mani87] Maniago, Pierette, "Consulting via Mail at Andrew," Large Installation System Administrators Workshop Proceedings, Philadelphia, PA, pp. 22-23, 1997.

[McNu93a] McNutt, Dinah, "Role-based System Administration or Who, What, Where, and How," Systems Administration (LISA VII) Conference, Monterey, CA, pp. 107-112, 1993.

[Ment93] Menter, E. Scott, "Managing the Mission Critical Environment," Systems Administration (LISA VII) Conference, Monterey, CA, pp. 81-86, 1993.

[Scha92a] Schafer, Peg, "Is Centralized System Administration the Answer?," Systems Administration (LISA VI) Conference, Long Beach, CA, pp. 55-61, 1992.

[Trocki] "mon" by Jimi Trocki, Service Monitoring Daemon, http://ftp.kernel.org/software/mon/ .

[Zwic90] Zwicky, Elizabeth D., Steve Simmons, and Ron Dalton, "Policy as a System Administration Tool," LISA IV Conference Proceedings, Colorado Springs, CO, pp. 115-124, 1990.

Author Information

Tom Limoncelli is a MTS at Bell Labs, the R&D unit of Lucent Technologies, where he is chiefly concerned with the architecture and operation of the data network for much of Research. Tom started doing system administration on VAX/VMS systems in 1987 and switched to UNIX in 1991, and in 1996 decided to focus on networks, not operating systems. He holds a B.A. in C.S. from Drew University, Madison, New Jersey. Reach him via U.S. Mail at Lucent Technologies, Room 2T-408, 600 Mountain Ave, PO Box 636, Murray Hill, NJ 07974-0636. Reach him electronically at <tal@lucent.com>. His web page is http://www.bell-labs.com/user/tal .

Adverse Termination Procedures -or- "How To Fire A System Administrator"

Matthew F. Ringel
Thomas A. Limoncelli – Lucent Technologies/Bell Labs

ABSTRACT

When an employee is terminated, his or her access to the organization's network and computer systems must be removed. However, the most difficult employee to terminate is often the person that built the system. We propose a three tier model for coordinating access removal that is useful in normal and adverse termination scenarios. We then work through a number of case studies to see how the model performs in this difficult situation. We feel this model performs extremely well. We also discuss, informally, how to minimize the risk of backdoors and how employees can reduce the possibility of being blamed for security incidents if they are terminated.

Introduction

One of the more unpleasant subjects in the working world is how to deal with a person who is being forcibly terminated from a company, possibly without warning. This could range from calling the person at home and telling them not to come into work again, to something as confrontational as two security guards meeting the employee at his cubicle, Human Resources informing him that his personal effects will be shipped to him, and the security guards escorting him out of the building. Either way, the outcome is a potentially unhappy ex-employee and, if the person in question is technically adept, a potential liability for the company's computer infrastructure.

This paper will discuss system administration practices that are used to reduce the risk of attack by former employees. Several companies have been interviewed to get an idea of the best current practices in the field. This paper will also consider what can be done to reduce the risk of backdoors that might have been installed by a former employee. In addition, this paper will also examine the terminated employee's perspective. It will offer advice on how to reduce or eliminate the possibility of being blamed for security incidents after you leave a company on not-so-amicable terms.

Motivation

Adverse termination tends to be looked at as a very rare exception, when in fact it happens more often than expected. No one likes to be forcibly terminated and, all kidding aside, there are very few system administrators who take pleasure in dealing with the termination and clean-up of recently departed employees. Therefore, termination procedures get about as much attention as one might expect for something deemed to be an unpleasant reality: ad hoc solutions when necessary, and not much in the way of formalized action. The authors note that larger organizations – corporate and academic – are much more likely to have termination procedures in place. With this in mind, this paper primarily focuses on small- to mid-size sites that are looking to go from ad-hoc solutions to a more formal approach. Larger organizations might find new ideas for looking at the complex issue of adverse termination.

The reasons why terminated employees need to be locked out of an institution's systems in an efficient and thorough way should be fairly obvious. What may not be so obvious are the measures and procedures that can be put in place to reduce the security risks created by a disgruntled ex-employee with access to your network. The purpose of this paper is to give some suggestions as to how one might establish a termination procedure, as well as some ways to put it into effect.

The Three Tier Model

The authors propose that the following three tier model should be used when constructing a procedure for removing access for any employee. The benefit of this model is that it can be used as the starting point when writing a formal policy for normal or adverse terminations, or used as a check-list for ad hoc situations. An example of an ad hoc situation would be a small company that does not see the need for a formal policy, but finds itself terminating a system administrator.

There are three aspects one must be concerned with when removing access: Physical access ("Can she get into the building?"), Remote access ("Can she remotely access our network?"), and Service access ("Have access rights to applications been withdrawn?"). This model is somewhat fail-safe because, as will be shown later, even if one of the three tiers isn't secured properly, the ex-employee will not be able to do damage.

Physical access means, quite simply, "Make sure they can't get in the building(s)." Usually this is the function of Human Resources or Corporate

Security. For example, Human Resources should already have a procedure for retrieving the employee's ID badge, which is checked by guards at the building entrance. If card-key access is used, the card-key system must be programmed to deny access to that card-key whether or not the card-key has been returned. Keys to any rooms must be retrieved or locks changed. Combination locks, safe combinations, etc. must be changed. If there are there multiple buildings, additional questions must be asked. Is there a barn, maintenance shed, shack, outhouse, dog house, or annex that must be considered? Because physical instruments like doors and keys have been around for longer than computers, most companies usually have good procedures related to them already. Follow them. Very small companies might not have any kind of identification badge or similar system in place. In that case, have the receptionist or other employees been notified of what to do if they see this person in or near the building?

Remote access refers to the many ways one might get into the networks. These include, but are not limited to, modem pools, ISDN lines, in-bound connections through a firewall, unfirewalled internet, and VPN service. Access to all of these systems must be disabled. This can be difficult if each of them is run by a different team or has a different access control system.

Service access refers to the applications and services that are inside the network. Each of these services usually has a password. Examples include IMAP4/POP servers, Database server, UNIX servers (each with their own /etc/passwd file to be checked, or a global NIS or Kerberos database to be checked), SMB servers (NT File Server), etc.

Logistically speaking, a team of people can be assigned to each tier. This three tier model is somewhat fail-safe, because if any single tier is not fully implemented but the other tiers are, then he or she will not be able to negatively affect the network. For example, if a UNIX account hasn't been disabled, but the ex-employee can't get into the network, he or she can't do any harm. If the ex-employee can get into the network via modem, but all services are disabled, the ex-employee will not be able to do damage (or will only be able to do the same damage internal employees are currently doing). This is where a single authentication database can save a lot of time. If all services are authenticated by NIS (or NT Domain, or SecureID) then there is usually only one database that must be updated to remove access. However, systems that use such a database often have "local" databases as well. UNIX hosts using NIS for password data still have an /etc/passwd file that is accessed before referring to NIS. NT has "local accounts." In this case, it can be useful to have an audit script that will seek out such local accounts.

Case Studies

Case Study #1: Large University – An Adverse Termination

Academic entities tend to be very astute about termination procedures, as they often have batches of terminated accounts at the end of every academic year. The university setting is a good example of how "practice makes perfect." However, our case study involves someone with privileged access to many machines.

At Large University, a long-time operator with root access to all UNIX systems was to be terminated. After some discussion it was decided to use the following procedure. A small team was assembled to list all the access she had and how to disable each, including card-key and host access. When the designated time arrived she was told her boss needed to speak with her in his office. Her office and her boss's office are in opposite parts of the building and the trip would take at least 10 minutes. By the time she reached her boss's office all of her accounts had been disabled.

Evaluation: Since Large University is a public university, they were not able to eliminate physical access to the building, but card-key changes prevented the person from gaining direct physical access to sensitive machines. Remote access could not be disabled since Large University does not use a firewall. Because the operator had access that was so extremely pervasive, the only way to assure complete coverage was to have all the senior system administrators involved in removing all Service access. Grade: A-

Case Study #2: Small Software Development Division (SSDD) – A Good, But Flawed, Termination

A system administrator left SSDD and offered one month's notice to help the company transition. The system administrator was actually involved in interviewing a replacement. All responsibilities had been transitioned the day before the termination date. As previously arranged, on his last day the system administrator walked into his boss's office and became root in a window on his workstation. While the boss watched, he disabled his own regular login. He then issued the command to change the password of various routers and let his boss enter the new password. He then repeated this for a few other "system accounts." Finally he issued the command to activate their "change root password on all systems" procedure and let his boss enter the new password. The boss was shocked that the soon-to-be ex-employee was doing such a complete job of transitioning duties and deleting himself from the system. Finally he turned in his card-key and physical keys and left the building. About two weeks later the ex-employee remembered that he had forgot to change the password on a particular non-privileged system account on a machine that had a modem directly attached to it. The ex-employee reports that he confirmed with former co-workers that

the ex-employer didn't disable the account until he notified them of the error.

Evaluation: With one exception, the termination was complete. However, the exception crossed Remote and Service tiers because the account (Service access) was on a host that was directly connected to the outside world (Remote access). Also, the authors do not feel it was safe for the new passwords to be set in front of the exiting employee, who could have been watching the keyboard. The employee also could have recorded the new passwords as they were being set. The garden-variety *script*(1) command does not record non-echoed input (such as passwords). However, a custom version of the *script*(1) command (let's call it my_script) that could record non-echoed input would make recording the passwords easy to implement.

This short script would allow the exiting employee to capture the changed passwords:

```
cd /tmp && my_script
Mail < typescript secret@example.net
rm typescript my_script;
```

The company took a significant risk by not disabling access as soon as notice was given. However, it was a reasonable risk to take considering that it was not an adverse termination. Grade: C-

Case Study #3: Large Hardware Company (LHC) Terminates The System Administrator's Boss – An Optimal Termination

LHC had to "suspend, pending investigation" the manager of a team of system administrators. As a result of the investigation, the employee was terminated. This person had administrative access to most systems in his domain and had access to the network via most of the remote access methods that the system administration group provided.

Without the accused manager knowing, a meeting was called by the accused manager's director, the lead system administrator, and corporate security. The corporate security officer explained the situation and the action plan. The lead system administrator was to change all the "root" (and privileged account) passwords that evening. In the morning, the system administration group (without their manager) would meet and be informed of the situation. They would suspend all access to the systems. If something could not be suspended, access would be deleted. The system administration group used an action plan similar to the authors' three tier model.

The lead system administrator spent the evening changing the root password on every system. In the morning, the system administrators were brought into a closed meeting and the situation was explained to them. The physical access aspects were taken care of by corporate security. The system administration team brainstormed on all the Remote and Service access issues. Breaking the problem down into two tiers focused their discussion.

One potential problem was the manager's home ISDN service. It would be 12 hours before the ISDN admin (a system administration team member currently on vacation) would return. The manager's home ISDN equipment had been surrendered, but it was possible that he could have programmed other ISDN equipment to enter the system. Even so, the risk of that happening was low, and the system administration team was confident that Remote and Service access had been sufficiently removed and logs could be monitored for intrusions. (In other words, doing a complete job on two of the tiers should compensate for being incomplete on the third tier.) Two hours later all access had been disabled.

The instructions to the team also included two important elements:

1. Keep a log of what is disabled and forward logs to the lead system administrator, who will maintain a master list.
2. Disable, do not delete. Out of respect for the manager, they must do all of this under the assumption that the manager will be cleared of charges. If and when the manager is exonerated, service will need to be restored immediately. Every service that they forget to re-enable will be a sad reminder of the entire ordeal. The system administration team did not want to have those painful reminders pop up months later.

During the investigation, the accused manager resigned. The logs were useful as a check list of what access had been suspended and now needed to be deleted.

Evaluation: This case study used the three tier model perfectly. Note the increased efficiency gained by splitting into teams, and how the inability to disable the home ISDN access was compensated by effectively removing all other access. Grade: A+

Case Study #4: Small Financial Company (SFC) Fires Their Lead System Administrator – A Worst-case Scenario

SFC was going to fire their lead system administrator due to non-performance. The lead system administrator had already been warned several times to improve his performance and knew that his termination was imminent. On the day before the termination, a junior system administrator (the only other system administrator in the company) was informed of the termination and given the task of revoking Remote and Service access for the lead system administrator by noon the next day.

The lead system administrator came into work the next morning and found that though he was able to log on to various administrative machines, his mail spool had been archived and deleted from live storage. Since the lead system administrator knew then that he would be terminated that day, and given that he had a considerable (and long-standing) grudge against the

company, he swung into action. Due to an ever-changing network layout, the lead system administrator had long ago planted a UNIX host of his own on one of the company networks without anyone noticing. The machine had been passively sniffing packets going across the internal network, capturing passwords over time. Once the lead system administrator knew that he was going to leave, he set the machine to a much more active mode, starting various services that would allow him (or anyone he gave access to) to get in from the outside.

As expected, two hours after arriving for work, two security guards and the lead system administrator's manager arrived at his desk, led him to a meeting room. Once there, he was told that he was fired, that he would be escorted off the premises immediately, and that his personal effects would be shipped to him. SFC was conscientious in making sure that he had no physical access to the property and posted security guards for the next week to assure that he wasn't going to try to physically break in.

Two months later, the remaining system administrator and the replacement for the lead system administrator noticed that they had an unusually large amount of traffic coming into their internal network from the outside world. As they started investigating that issue, the Chief Technical Officer called the system administrators, informing them that SFC's web pages had been defaced, and that the company was fielding inquiries as to whether or not SFC had been hacked. The new lead system administrator verified that the company's web pages had been defaced, noted that there was a large amount of data flowing out of the site, and went into the machine room and pulled the plug on the router connecting SFC to the Internet.

The next several weeks were spent doing damage control (both network and corporate image), figuring out when the intrusion happened and how much damage had really been done, and then re-installing operating system and application software from media. Eventually, the system administrators found the rogue machine, completely devoid of data (no doubt it had erased itself when it was cut off from the world for an extended period of time). Unfortunately for the company, the first signs of intrusion occurred about a month before they noticed the upswing in traffic. The swell in traffic was correlated to a message posted in a Usenet newsgroup, saying that the following system was "owned," along with directions on how to access it. The system administrators hypothesized that the former lead system administrator has entered and compromised the network remotely about a month before the Usenet posting; these were the first signs of intrusion they found. The Usenet posting, as well as the self-destruction of the rogue machine, were meant to cover the original infiltration.

Evaluation: Although physical access was secured remarkably well, the Remote and Service access tiers were secured haphazardly or not at all. No firewall will protect you against an internal attacker, especially one who is mostly passive (i.e., the rogue machine). Grade: F (although an honorable mention is given to the replacement lead system administrator for understanding that when feasible, pulling the plug is one of the most effective ways to secure a network).

Checklists

We have assembled several checklists of things to disable in the event of an adverse termination. While no checklist is complete, these can serve as a basis for your own procedures. You might want to use your company's "new hire" procedure as a starting point. In general, whatever is done for a new hire has to be undone for a termination.

- Physical access (Tier 1):
  - Change combination locks
  - Change all applicable safe combinations
  - Doors with keys should have locks changed (even if keys are returned)
  - Have ex-employee surrender:
    1. keys
    2. card-keys
    3. hand-held authenticator cards
    4. PDAs (Pilot, Newton, etc.)
  - Remove access for all buildings (e.g., remote locations, shacks, utility buildings, etc.)
- Remote access (Tier 2):
  - Modem pools
  - ISDN pool
  - VPN servers
  - In-bound network access (i.e., ssh, telnet, rlogin)
  - Cable Modem access
  - xDSL
  - X.25 access
- Service access (Tier 3):
  - Database servers
  - NIS domains
  - NT domains
  - Netnews: "news," "Usenet," and "uucp" passwords
  - RADIUS servers

Improving Accuracy

A good inventory of possible access points improves your accuracy. If these case-study sites had had well-managed inventories, they wouldn't have had to do as much last-minute brainstorming.

A good way to improve accuracy is to reduce the number of access databases. Large numbers of access databases (for example, an /etc/passwd file on every machine, embedded passwords in routers, etc.) increases the amount of work needed to remove access and increases the chance that coverage will be incomplete. If all access was controlled by a single database (impossible in today's environment), all access could

be disabled from a single point. In case study #3, access to many services (VPNs, in-bound telnet through the firewall, root access, etc.) were controlled by hand-held authenticators (HHA) keyed to a particular database. Disabling the manager's record in the HHA database resulted in immediately disabling many different services at the same time.

Preventing Backdoors and Logic Bombs

We have all heard a variation on the story of a disgruntled employee who sets up a program at work that he must deactivate every week or it will cause massive damage within the company's system infrastructure. Sure enough, the person is terminated, and the program "goes off," wrecking two months of work by everyone in the company (or something similar). While the story itself might be apocryphal, it is entirely possible to build such a program. In addition, there's the story of the terminated employee who kept a modem hooked up to a spare analog line in the data center, and was able to access various internal systems from the outside, completely undetected by any intrusion detection infrastructure. This story is also apocryphal by itself, but is an entirely likely scenario.

The first story included an example of a "logic bomb": an autonomous program that is set to trigger a detrimental event under some set of conditions that is known only to the writer. The second is an example of a "backdoor": a covert way of entering a system or network while bypassing security that was installed after the system went live. Either one could be planted by someone who is leaving the company in the hopes of getting revenge at some significantly later date. If the perpetrator has administrative access, there are any number of ways to hide a logic bomb or a backdoor, but there are ways to increase your chances of finding or eliminating them before they do their dirty work.

One possible method of sweeping for logic bombs or backdoors is to compare each machine to which the person had administrative access (a "touched" machine) with another machine that has a known-good "/" and "/usr" filesystem. Ideally, you will have taken a snapshot of the checksums when the machine was put into production (using tripwire [1] or a similar tool) and placed it onto read-only media in a safe place. The question of how to keep that media safe is beyond the scope of this paper, although one of the authors saw a bank safe-deposit box used to great effect for that purpose at a previous job. Maintaining checksums of "/" and "/usr" for each production machine can be cumbersome, but is not hard to automate and will save you trouble in the long run.

Next, check the *crontab*(1) and *at*(1j) files for every user on each touched machine. Every job must be accounted for. A machine might appear to be running perfectly, but you don't know what the former employee might have inserted into those files, especially if they're large enough (e.g., root's *crontab*(1)) that no one really holds them up to close scrutiny.

After that, you should probably make a thorough accounting of all the daemons running on each touched machine. Much like *cron*(1) and *at*(1), there is usually so many programs running on any given production machine that yet another daemon will get lost in the noise.

Checking the checksums, cronjobs, and atjobs on each touched machine is a very good first step to show that a machine is clean from the most blatant and simple attacks. It should be noted that it is not difficult for a skilled person to hide themselves extremely well. For the cautious, one should put a statically-linked copy of *ps*(1) on the machine, and check for daemons using the statically-linked version of the command. The same goes for a statically-linked copy of *crontab*(1) and *more*(1) when examining other files on the system, just so you know you're running commands with uncorrupted libraries.

One last thing to check for is machines that are passively sniffing packets on your network. As mentioned in Case #4, no firewall will stop an internal attack, and a machine that passively collects data and stores it for later transmission or retrieval is one of the more straightforward and effective backdoors. As of this writing, there is a publicly available tool called AntiSniff [2], which uses a combination of OS-specific and protocol-based transmissions over Ethernet to determine whether a machine is passively sniffing packets from the network.

All the advice given above is for the purpose of maximizing uptime while doing your audit. There will be times when the terminated employee in question is highly skilled and will be able to hide themselves from even a reasonably close inspection of a system. Sometimes taking a system down and re-installing from known-good media is the only choice, such as in Case #4.

A full backdoor audit checklist is outside the scope of this paper. The authors recommend the "Server Security Checklist Overview" [3] and "UNIX Configuration Guidelines" [4] as papers that investigate a large variety of typical system installation pitfalls which may be used as backdoors, and how to protect against them.

The Other Side of the Coin

Now that we have taken a look at the employer's side of the equation, we will talk about the employee's side: what to do when you're the one going out the door, and not necessarily under friendly circumstances.

The authors would like to note that they are not lawyers, and that the law is often ambiguous in this area, mostly due to lack of court precedent. The only

advice that we can give with confidence is that if you are terminated, immediately remove your hands from the keyboard and never touch a computer of your employer's again. That, plus a good lawyer, is the best (albeit possibly flawed) advice that we can offer.

If you are being terminated unilaterally for whatever reason, and you've had administrator access on any machine that is considered important, it is safe to assume that your soon-to-be-former company considers you a security risk, as you are not only knowledgeable, but possibly disgruntled. Therefore, you want to make sure that the company has absolutely no reason to believe that you are responsible for any computer security mishap that happens in the future.

There are two scenarios to consider:

- When you see the termination coming.
- When you don't.

It is beyond the scope of this paper to discuss ways to tell that your termination is imminent. It's not hard to figure out when your employer is planning for your departure, even if your employer hasn't told you yet. If you see termination in your future, here is a list of things that you can do to make sure that you leave in good faith, if not on the best of terms.

- Make a list of the machines to which you've ever had administrative access. As a system administrator, your employer will assume the worst. He might suspect that you have planted logic bombs, especially on machines that made their way into the data center without much administrative scrutiny. Volunteering all the information you can is your first priority. It is a double-edged sword, though. If you miss a single machine, you open yourself up to being accused of hiding something.
- Be civil. It sounds obvious, but when you're getting terminated against your will, there will be hostility and you want to make sure things go as smoothly as possible.
- Set personal files aside. The authors believe that personal and work files should stay completely separate, but accidents happen. Assume that your employer will go through all of your files, including your voicemail and mail spool. Set aside a well-marked directory with your personal files, and mention it to your superior when the time comes. If you are up-front about the information, you're much more likely to get it all back intact. Do not be surprised if they do not permit access to the data, though. Make sure you understand any and all relevant policies. If your employer forbids the storage of personal data on their computers and you have personal data clearly marked and set aside, your employer has further grounds for termination.
- If you haven't already, make professional connections with your fellow employees. The people you work with will be the best defense of your character if your superiors suspect foul play by you. They are also a good source of references for your future job search.

When you don't see the termination coming and you're pulled into a meeting similar to the person at the Large University mentioned in Case #1, the best you can do is volunteer as much information as possible. Give all of your passwords, be very businesslike, make sure that you get what's coming to you (i.e., vacation pay, severance, other benefits), and walk out the door without a fight. It might hurt your pride, but in the end it will save your professional reputation.

DO's and DON'Ts

There are a few rules of thumb that can be drawn from the technical and non-technical recommendations mentioned in this paper.

For the Employer

DO the job quickly. Once the process starts, don't let it drag on. If you do, you will have a demoralized employee on your payroll.

DO make sure that you can effectively "flip the switch." Make sure that you and your staff are ready to remove all access at a specific time. As an example, having email access go away a few hours before account access will not only look unprofessional, but tip off the employee, especially if they are hostile.

DO make sure that all the necessary staff is informed of the termination. No more, and (more importantly) no less.

DO prepare to re-install the operating system on any touched machine. You can lock the person out from the machines, but their programs might still survive. As a good first check, you can check the crontab and atjobs to make sure that everything running has a well-known purpose.

DON'T get cute or clever in finding a way to communicate to the person that they are being terminated. This is not a game, and your employee is a human being with emotions. Your Human Resources department will have specific policies about how the communication is to be done.

DON'T treat the employee like a criminal. Mutual civility is important. The person was hired as a professional and you must respect that status during termination proceedings. Besides, you might be looking for a job someday, and this employee might be part of the hiring process.

For the Employee

DO make sure that all your project work and documentation is in order. Actively making the transition easier goes a long way to reassuring people that you're leaving the company in good faith.

DO volunteer all the information you can about what was in your control, the status of your projects, and all of your passwords. Once again, this is all about leaving in good faith.

DO resist the urge to give a "parting shot" by sending out email to all of your coworkers that you're leaving and "going to a better place." It might make your former employer question your professionalism, which might cause them to see you as a security risk.

DON'T mix work and personal files. You will eventually leave the company at which you're currently working and separating the two gets harder with time. In addition, your employer probably has a policy about keeping personal data on corporate computers. Usually, the policy forbids it. Even if you have a more liberal policy, it may be a good idea to adopt a personal policy that is more strict.

Conclusion

Securing systems after employees have been terminated is a problem that is not given much attention because of the unpleasantness of the subject. This results in ad-hoc solutions that are of varying effectiveness. The authors present a more formalized model. The process of securing systems from an employee who has just been terminated can be broken down into three tiers: Physical access, Remote access, and Service access. The model is fault-tolerant and is still effective if one of the three tiers is not secured properly.

The primary benefit of the three tier model is that it provides a structured approach to securing a network against a knowledgeable attacker, with the additional benefit of providing staff with a way to look at securing the network from other kinds of attackers as well.

This model is one of many different approaches to the issue of securing systems against terminated employees. The authors hope that this paper will encourage others to give thought to this often overlooked and difficult subject.

Author Information

Matthew F. Ringel received a BS-CS from Columbia University in the City of New York in 1995, after which he worked as a system administrator for Phantom Access Technologies and PSInet/Pipeline New York. He currently lives in Cambridge, Massachusetts and is a network administrator whose current projects include process design for a network operations center and large-scale implementation of several network management systems to monitor a major web-hosting provider. Reach him electronically at ringel@mithril.org.

Tom Limoncelli is a MTS at Bell Labs, the R&D unit of Lucent Technologies, where he is chiefly concerned with the architecture and operation of the data network for much of Research. Tom started doing system administration on VAX/VMS systems in 1987 and switched to UNIX in 1991, and in 1996 decided to focus on networks, not operating systems. He holds a B.A. in C.S. from Drew University, Madison, New Jersey. Reach him via U.S. Mail at Lucent Technologies, Room 2T-408, 600 Mountain Ave, PO Box 636, Murray Hill, NJ 07974-0636. Reach him electronically at <tal@lucent.com>. His web page is http://www.bell-labs.com/user/tal.

References

[1] Tripwire Security Systems *The Tripwire Intrusion Detection System*, http://www.tripwiresecurity. com/prodintro.html.

[2] L0pht Heavy Industries *The Goal and Purpose of AntiSniff*, http://www.l0pht.com/antisniff/purpose.html.

[3] Vandenberg, P. D. J. and Wyess, S. *Securing Solaris Servers – A Checklist Approach,* Feb. 1999, http://cne.gsfc.nasa.gov/security/sun_security/index.html.

[4] CERT Coordination Center *UNIX Configuration guidelines,* Oct. 1997, ftp://info.cert.org/pub/tech_tips/UNIX_configuration_guidelines.

Organizing the Chaos: Managing Request Tickets in a Large Environment

Steve Willoughby – Intel Corporation

ABSTRACT

The Intel Performance Microprocessor Division Engineering Computing (PMD EC) team consists of 85 systems administrators serving the needs of 2,000 engineers while trying to manage 7+ terabytes of data and 2,500-3,000 computers. Without some kind of task management strategy, the needs of so large a group would quickly spread chaos and confusion. In fact, due to the nature of our relationship with our "customers" (PMD chip designers and other related employees and contractors), who have strict and demanding requirements of their computing environment's features and stability, extra discipline is required just to keep on top of our issues, and to properly prioritize and execute them.

The EC group receives several hundred customer help requests each week. To organize and track these requests, ensuring that they are done in the right order, none are forgotten, and that our customers are kept informed as to our progress, we have evolved a set of tools, collectively named "REQADM" (for "REQuest ADMinistration"). This tool suite allows customers to submit requests, which get assigned to EC technicians to be worked on, and eventually brought to a resolved state.

This paper describes the REQADM tools, the organization we evolved to support our customers, and the lessons learned along the way.

Pre-History

We started with a system similar to Trouble-MH [3], QMH [7], Req [2], QMH-with-complainers [9], and Request [8]. We set up E-mail aliases with names like "sys-help" and "hot-help" (the latter for requests deemed – by the customer – to be more urgent than normal). These aliases automatically dumped messages to MH-style archives, thereby effectively assigning a number to the "request ticket".

On the way into the archive, the ticket got stamped with some administrative information at the bottom of the message, which looked something like this:

```
XXX Admin Information Follows XXX
NUMBER:     23669
DIRECTORY:  sys-help
RECEIVED:   Fri Feb 9 11:00:58 1996
```

The file itself remained in RFC 822 [1] format, with mail headers intact and the request text and administrative information as the mail message body.

An EC administrator (called the "request dispatcher") would have the task of monitoring the incoming "queue" of requests. This would be done by running, as often as possible, a Perl script to search all the files in the *sys-help* and *hot-help* directories, scanning their contents to see if any had been assigned to an owner yet, and printing out a list of any which had not.

From this list, the dispatcher would run a Perl script called *assignrequest*, assigning each new ticket to an EC administrator who would "own" the ticket and be responsible for seeing it to completion. This would cause a set of text lines to be appended to the ticket file:

```
ASSIGNED TO:      steve
ASSIGNMENT DATE:  Fri Feb 9 ...
ASSIGNED BY:      vicki
```

Now, the user "steve" would be responsible for the ticket. An E-mail notification would be sent by *assignrequest* to the original submitter and the new owner.

There was a large set of shell and Perl scripts used to manage the ticket in their MH-format files: *noterequest* to add a note (status update, commentary or question), *reassignrequest* to change the owner, *request.nag* to send nasty notes to EC admins who had requests getting old, *showrequest* to view the contents of a request, *whoserequests* to scan the queues to see who owned what requests, and so forth.

Problems With the Old System

We ran into a few problems over time, which made us think we should reconsider how we organize our team and our ticket tracking software.

First of all, since the files were nothing but MH mail archives, they were vulnerable to a host of mishaps. Bugs crept into the scripts as we added hack upon hack, some of which corrupted our ticket files. For example, if a script had to reassign a ticket to a new owner, it had to add "ASSIGNED TO:" lines to the bottom of the file (no lines were ever deleted, so we always had a record of what happened to a ticket):

```
ASSIGNED TO:       fred
ASSIGNMENT DATE:   Fri Feb 9 ...
ASSIGNED BY:       steve
NOTE:
I'm assigning this to Fred, since
I'll be out of the office for a
few days. He'll be taking over
for me.
```

Since our scripts relied on being able to use *grep*(1) to search for keywords like "ASSIGNED TO" to see who the current owner was, the new script had to know to munge the previous assignment(s), so they wouldn't be mistaken for the new one:

```
ORIG-ASSIGNED TO:      steve
ORIG-ASSIGNMENT DATE:  Fri Feb 9 ...
ORIG-ASSIGNED BY:      vicki
```

As the system features expanded, and the number of management scripts multiplied, it was inevitable that a new script would corrupt data that an old script assumed would be in a certain format.

There was also the risk that user data or admin comments (see the "NOTE" added to the re-assignment above) would contain text which would look like an administrative keyword like "ASSIGNED" or "CLOSED". As new keywords were added by new scripts, old tickets which didn't break before suddenly caused problems.

There was little possibility for control over who was allowed to edit tickets, since everyone in the EC team had to have write access to the files to run MH commands and ticket-management scripts on them.

And of course, every so often, someone would forget that he or she was in their ticket "folder" and issue an MH command to re-pack the folder, effectively renumbering all the tickets in the system (duplicating numbers of older closed tickets). At other times, the MH sequence files would get corrupted and all incoming requests would end up with ticket #0 (effectively going nowhere).

The scripts were getting fragile as their interdependencies (and even their own internal code) became more convoluted and difficult to maintain. The old system was collapsing onto itself, and needed serious redesign.

Customer Relations Issues

Some of the problems we faced with this previous model were not completely related to the software tools being used. No computing support group operates in a vacuum, and the way we interact with our customers is an important factor to be considered in evaluating and redesigning a support system.

For one thing, it can be advantageous to have one's ticket queue visible to the user community, as I discovered at the company I worked for prior to Intel, where I was the sole systems administrator. I implemented a simple ticket tracking system there to help my users understand how many issues I was dealing with at any given time. This made them far more sympathetic to the fact that each of their requests were to be prioritized and some would have to be denied or deferred for lack of available time to spend on them.

On the other hand, this same feature is a bit of a double-edged sword. We are also aware that whatever problem is preventing a customer from getting their work done is extremely urgent to them, and it may not seem relevant at that moment that there are other problems elsewhere in the department that also require EC's attention. This can be frustrating to all concerned. We wanted to make sure that the ticket tracking system we migrated to would help our customers see where their issues fit in to the bigger picture. This would help them understand what priorities we need to juggle for the common benefit of all. It would also help EC be constantly aware of what the "hot" issues are, to be sure we are always working on the right issues in the right order.

One of the problems with our MH-based system, which made this situation difficult to resolve, was the fact that the system only kept a single bit of resolution for ticket priority: it was either "normal" or "hot". Since this did not provide us with enough insight to adequately judge relative priority between tickets, there were times when EC technicians would mis-prioritize jobs, frustrating our customers, and times when customers would file too many requests as "hot", diluting the meaning of that priority.

This is actually a combination of technical and political (managerial) issues, and we approached solutions along both of these lines, as we'll discuss below. From a technical standpoint, however, one thing that was clear to us was that any new software solution needed to include features to give the customer more control to specify true and accurate priority.

Another problem we faced was that there was some concern that tickets were occasionally closed before the problems they described were fully resolved. This was not a major problem for us, but it is the kind of thing that doesn't need to happen very often to become a customer relations issue (particularly if it happens to a key customer).

While there may have been isolated instances of less-than-careful work on the part of a sysadmin, or a customer feeling uncomfortable with the actual solution offered, the chief source of concern here was communication between EC and the customer. Most often, when this happened, the real issue was simply that the sysadmin didn't understand exactly what the customer needed, or the customer didn't fully understand the nature of the problem, and therefore that the solution offered would really solve it.

This pointed out to us that we needed to implement a reliable mechanism within our new system which would formalize – at least – a communication loop for final approval of a problem's resolution to be certain that all was resolved to everyone's satisfaction.

Some Attempted Improvements

While the new request system was being designed and written, the computing group made a number of other attempts to improve our situation. These included public relations, process, organization and software improvements.

Customer Surveys

It is important to the success of a support organization to know how well they are perceived by their customers to be meeting their needs. The best way we are aware of to get this information is to ask them for feedback.

This is not as straightforward as it sounds, however. Our customers, being designers and engineers, are usually very busy at work inventing and improving products for our company. We want to take up as little of their time as possible to perform "overhead" paperwork like customer surveys. (We would also like to take up as little of our time as possible to administer and analyze these surveys as well.)

The old software had no automated features for polling customers to see how they felt about our service, so our attempts to assess their satisfaction level were manual and occurred at irregular intervals.

Initially, we tried the "broadcast spam" method, sending a general survey by E-mail to everyone we supported, asking them to reply with the answers to our questions. Only about 5% of our customers responded to the surveys, and most of them were our more vocal users, from whom we were already hearing feedback anyway.

Later, we tried the "barbarian invasion" method. One of our key front-line support people would randomly select a few tickets submitted each week, and go sit down in the submitter's cubicle for five minutes and interview them to get their feelings about how well their issue was handled. This was more costly (in terms of time spent) to implement, but ultimately was more successful in gathering feedback from a wider sample of our user population.

We eventually decided that our new tracking system would need to incorporate some capability to poll customers who are using it, asking them a few questions about how well their needs were met. (As of this writing, these features are being designed for the next release of REQADM, which should be finished by the time this paper is actually published and presented.)

New Queuing Theory

While we were designing our new request system, there were a number of new ideas proposed and considered. One of the more interesting ones involved changing our fundamental queue-management philosophy. The hypothesis was that our "multiple queue, multiple server" system was inefficient, since every "server" (i.e., every EC support technician) had his or her own queue of tickets assigned to be worked on. In some cases, some tickets in a system administrator's personal queue would wait for other tasks to be completed, when another individual might have been able to take them on in the mean time. We can liken this to a grocery store check-out line, where you have to pick a line and wait in it, although a customer ahead of you may take longer than you expect, but you can't just jump over to a faster line without going to the back of the line again.

The new system proposed was a "single queue, multiple server" system. In this system, which we can liken to a bank teller queue, all the incoming tickets would wait in a single queue until a support person was actually ready to work on it. The theory was that tickets would move faster through the system.

We implemented a script called *getrequest*, which support people would use to get the next available ticket from the incoming queue, and assign it to themselves. They would then work on that ticket until it was finished, and run *getrequest* again.

There were, however, a number of concerns we had about basing our entire ticket-handling methodology on this new concept.

First of all, there will always be a number of tickets which must wait for some external event, such as a part arriving from a vendor, or a customer response, and while that ticket is "on hold," the next ticket on the list is grabbed and started. Before long, individual support members would build up their own queues of tickets again anyway, and eventually, we reasoned, they would stop taking incoming tickets while they tried to clear their personal ticket backlog.

Secondly, we have to recognize that systems administrators are human beings. We wanted to be very careful not to establish a ticket-handling system which relied too much on diligence to manually get as many new tickets as one could get. We were concerned that there would be too much temptation to delay grabbing a request if a difficult or unpleasant request is next in line.

Thirdly, along the same lines of human nature issues, there are a lot of other tasks systems administrators need to concern themselves with every day, so we feared that a natural inclination would be not to get tickets as often as possible, on the basis that we each know we are already quite busy with other important tasks and don't have free time to take on a ticket yet. In our original system, where tickets were dispatched to support people directly, there was more of a tendency to just accept the workload and work all the tickets in to the day somehow, so more work was actually accomplished.

However, there were still some interesting uses for the techniques we discussed for this queuing theory, and this did affect how we assign requests into certain "rotation" groups, as we'll discuss below.

Improved Queue Displays

An artifact of the old system was that a lot of ASCII files needed to be opened, parsed and scanned

in order to view what was in the incoming queue of tickets (for the ticket dispatcher), as well as an individual's personal queue.

Originally, this was done by a simple shell script which everyone would run every few minutes, rescanning all files every time. An improved version of this script was implemented which ran continuously in an xterm window. It would make a pass through the queue directories every few minutes, but would only parse and scan files modified since the last scan. Then the results were displayed in a sorted, color-coded table, helping a support person locate information quickly and easily.

Automatic Request Dispatching

To relieve the heavy burden of assigning tickets to suitable people without overloading them, an automatic system called "REQMGR" was created.

General areas of issue ownership were identified, and fake usernames created for them with names like "postmaster" (for mail issues), "operator" (for backups and restores), "unixreq" (for general UNIX issues), "ntreq" (for general Windows NT issues), etc. These owner names were called "rotations", since at any given time, some subset of our support group would be "on duty" for their shift on one of those areas as part of a rotating schedule.

The REQMGR script runs every five minutes out of *cron*(8) and scans the active queues for any tickets which the dispatcher assigned to a "rotation". For each of those, REQMGR looks at the rotation schedules to see who the current candidates are, and assigns the ticket to the candidate who is the least loaded with other tickets.

This has proven to be a great asset, and has been integrated into the new REQADM system still in use today.

Looking in Other Directions

At this point we were facing the prospect of having to "reinvent the wheel" by redesigning our entire ticket system. Before going to those lengths, we tried to investigate other products which might already work for us.

Remedy AR System

One option was the Action Request System by Remedy. This was (and still is) a tool being successfully used at other Intel sites, and was worth looking at for our purposes. We set up a Remedy server, dedicated a full-time person to configure and administrate it, and tried to move our staff and customers over to the AR System.

While AR System is itself a fine product and many sites are happy using it, we found that it didn't quite fit into the established way we preferred to handle tickets.

From our point of view, we needed a system we could customize heavily to our department's needs and preferences. Further, any deviation in the general approach to tracking tickets from the old system to the new one would be a significant stumbling block. Our customers rightly argued that as they were on tight development deadlines, a substantial change to the process by which they entered their tickets, and EC's process for resolving them, would impact their overall productivity and might slip project schedules.

One of the main features our customers absolutely required was the ability to send free-form text messages via E-mail to a "sys-help" alias, and have that message automatically enter the ticket system database. They wanted updates to be mailed to them every time an action was taken on their ticket. This further justified a different solution where we could customize all these interfaces to our liking. (AR System does allow mailed requests, but they need to adhere to a fairly specific format.)

Helpdesk Institute Conference

We also sent a contingent of our support staff to a conference sponsored by the Help Desk Institute [5]. They attended talks and workshops specifically oriented to running successful helpdesk centers.

They returned with a number of interesting new perspectives on help desk center management and structure, which influenced the redesign of our "Computing Support Center". They also returned with a report that the conference attendees who were the most satisfied with their request ticket management software were the ones who designed their own custom systems.

This helped us to justify going ahead with the design and implementation of our own custom system.

This sentiment has been echoed by many others in the industry, including D. Johnson of Network Computing Group, who stated, after describing a model for "ideal" trouble ticket systems, "It is hard to imagine that this whole system could come out of a shrink-wrapped box..." While mentioning that commercial RDBMS packages' screen form and report generation facilities may be a good start, he found it "difficult to integrate full trouble ticket functionality through these systems [6]."

Design of a New System

We assembled a team, composed of technical contributors and management from the EC group, to work out the design of the new system. Over a period of several weeks of brainstorming, discussion, and only occasional arguments, the following system was sketched out, and development began.

In all of this, our goal was to design a trouble ticket system suited specifically to our needs in Engineering Computing, optimized to our specific support and business models.

It came as a pleasant surprise when we later found RFC 1297, *NOC Internal Integrated Trouble*

Ticket System Functional Specification Wish-list [6] and compared it against our REQADM system. The system we developed independently matches almost the entire wish-list in the RFC. To some degree, this gave us a feeling of validation that we must at least generally be on the right track with REQADM. It also validates the RFC, in that it accurately portrays a large support organization's ticket tracking needs.

New Computing Support Structure

Part of our efforts to revamp our customer support organization involved changing the way we ran the support center.

Originally, our support model was a "free-for-all" affair, where all tickets submitted by customers were directly assigned to an appropriate systems administrator. This was frustrating to EC and our customers alike, since it tended to overload the sysadmins who already had full workloads of their own, in addition to carrying a backlog of one or two dozen tickets (which just had to be "worked in" to the day somehow).

We looked at the way computing centers were being run elsewhere in our company, took the best of the ideas we found from each group, and created a system that would work well for our specific needs.

We created a "Monitoring and Control Center" (or "MCC") which would use automated network and host monitoring tools to proactively watch for trouble, fixing problems *before* customers started complaining about them.

The name of the center was later changed to the Computing Support Center ("CSC"), because some of our customers thought the "monitoring" and "control" words referred to our customers rather than their systems, and thought it sounded too Orwellian for their comfort.

To relieve much of the burden of ticket handling from the bulk of our systems administration personnel, we staffed the CSC with a full-time crew to be our "front line" tier, answering phones and handling all the tickets they could. This immediately removed all tickets not requiring heavily specialized expertise from the majority of sysadmins. Eventually, the CSC was able to handle more and more kinds of tickets directly, as area owners were able to train the front line to handle common problems and procedures such as new account creation and mailing list administration.

To supplement the front line crew, everyone else in EC (up to and including senior management) agreed to serve a week-long "rotation" in the CSC. During this week, they do nothing at all except handle request tickets that come in, for which the front line people aren't trained to handle, such as specific technical areas of UNIX or NT administration. By having them in the same physical room (the CSC), everyone on duty that week maintains a high level of communication about the current issues being dealt with. As an additional benefit, the full-time front line people (who are typically our newer, less-experienced administrators) get hands-on experience watching and helping the more seasoned administrators handle these issues. Many of the front-line crew eventually get moved on to specific systems administration positions based on their experience gained in the CSC.

To offset the stress and turmoil of dealing 100% with user requests and telephone calls for a solid week, once a sysadmin's rotation week is completed, they get to leave any unresolved tickets behind for the next week's crew to continue working on, and they don't have to return for ten more weeks. So the next ten weeks can be as close to "uninterrupted" as possible for any systems administrator,[1] and more work on projects and other administrative work can be accomplished than was possible when everyone was carrying that burden all the time.

Life Cycle of a Request Ticket

Requests for work are initiated in a variety of ways. Most commonly, a customer will have a question or will notice that there is something wrong in their work environment. They will use one of the interfaces described below to communicate their issue, directly entering a new ticket into the REQADM database.

Alternatively, they may call our CSC hotline and explain their problem to a front-line technician. Unless it's a trivial question which can just be answered in a couple of sentences, the front-line tech will call up a blank ticket form on their screen and enter a request on behalf of the customer.

Sometimes, however, one of our myriad system-monitoring scripts discovers a problem and either solves it directly or generates a REQADM ticket for a system administrator to look at.

Once the ticket enters the REQADM system, by whatever means, it has a "new" status, which means that it has newly arrived and has not yet been assigned to a system administrator.

Birth of a Request Ticket

Ideally, we want to prompt the user for specific information when they submit a ticket into the system. This has taken a fair bit of work, since in our old system customers would simply E-mail a free-form text message to a designated mail address. The problem with such unstructured input is evident when you receive request tickets which look like:

```
From: j_random@ichips.intel.com
To: sys-help@ichips.intel.com
Subject: FIX IMMEDIATELY

My environment is broken.
Pls fix now.
```

[1] In other words, not much, but it's definitely an improvement.

Naturally, immediately after submitting the ticket, the user leaves the office for the day, leaving the EC support group at a loss to even understand the complaint.

From the customer's point of view, however, the ease and simplicity of sending a simple mail message to initiate a ticket is very attractive, so our new system had to continue to support this.

However, the new system includes interfaces which we advertise as the "preferred" method for contacting us. The most basic is to simply type *request* at a shell prompt. This brings up a data-entry tool as shown in Figure 1.

To avoid overwhelming the user with too many input fields, the entry form is tabbed "notebook" style. The first tab, which identifies the user and provides us with contact information, is totally pre-populated by looking the user up in our corporate employee database. Most users need only check this page for accuracy and move on.

The second tab identifies the specific system being complained about. This section is only needed for workstation-related tickets and can be ignored when not applicable.

The next tab allows us to specify the attributes for this ticket. The request queue is one way we can organize types of tickets. The vast majority of tickets submitted go in the default "request" queue. We have defined other queues for special purpose situations, such as requests from EC to outside vendors, long-term projects, TO-DO lists, and so forth. Each queue has its own rules for agreed-upon resolution timeframes, and can be separately queried for reports.

The other two fields are of the greatest importance to the customer submitting the ticket. In the old system, there was but one way to indicate priority. You could mail your request to 'sys-help' or to 'hot-help.' Having only one bit of resolution for ticket priority proved woefully inadequate. For example, there was no way to distinguish "this task is extremely urgent and the world will end if it's not done in two hours" from "this task can wait until Friday but is extremely important that it really be completed at that point, or the world will still end."

To solve this, we prompt for two separate data points. The first is a four-level "importance" rating:

Urgent: Emergency situation; multiple users are down; all work stopped.

High: Significant impact to work being done; resolution needed ASAP.

Medium: Default priority – some impact to work; other work may still progress normally; resolution needed when reasonably possible.

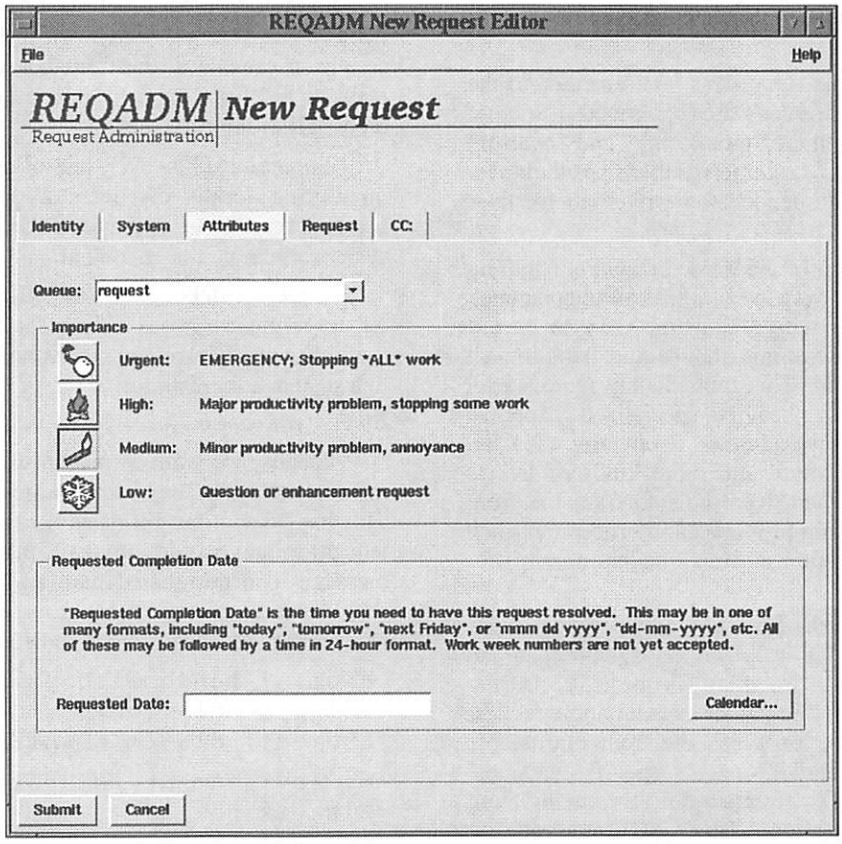


Figure 1: Request data-entry tool ("attributes" tab).

Low: Not very important; request for routine enhancement; random questions.

This indicates how much the issue is impacting the customer – how important it is to be resolved – without explicit regard to a deadline for completion.

The second data point is a requested due date. This indicates an explicit point in time when the customer requires completion of the request. If none is specified, a default deadline is calculated based on importance rating and our contractual SLA with the customers. (If the customer requests a deadline earlier than our SLA contract has agreed to provide, it's not considered binding, but it's still useful to know.)

With these two data points, we have a much more accurate indication of how best to prioritize this ticket in the list of other tickets we're dealing with.

The next tab allows the customer to enter free-form text describing the problem in detail (see Figure 2). To help inspire the customer to specifically mention how this problem might be reproduced in the lab, we explicitly ask for this on the form.

The input tool can be customized to add special fields to this tab as appropriate for specific problem types. For example, if a customer is requesting additional disk space, it may add fields asking how much space is needed, on what fileserver, for what group, etc. Or, if access to a UNIX group is requested, it could look up the groups the user currently belongs to and set up a menu of other groups to select from.

The last tab allows us to specify a list of other people to be copied on all mail correspondence regarding this ticket.

Once we are finished entering the relevant data, getting the ticket into the system is only a click away. REQADM gives us immediate confirmation with the ticket number for our future reference.

Assignment

Meanwhile, back in the CSC, a front-line technician is on duty to watch for, and dispatch, incoming tickets. On his REQADM display, new tickets appear highlighted at the top of the queue list; see Figure 3.

These tickets can be opened directly from this list, and examined or edited by the support technician; see Figure 4.

From the type of problem in the complaint, the dispatcher assigns the ticket to another support technician. For example, if the request were about a problem with a mailing list, it might be assigned to "unixreq". This is the standard rotation for UNIX requests. This will then automatically be assigned to the next

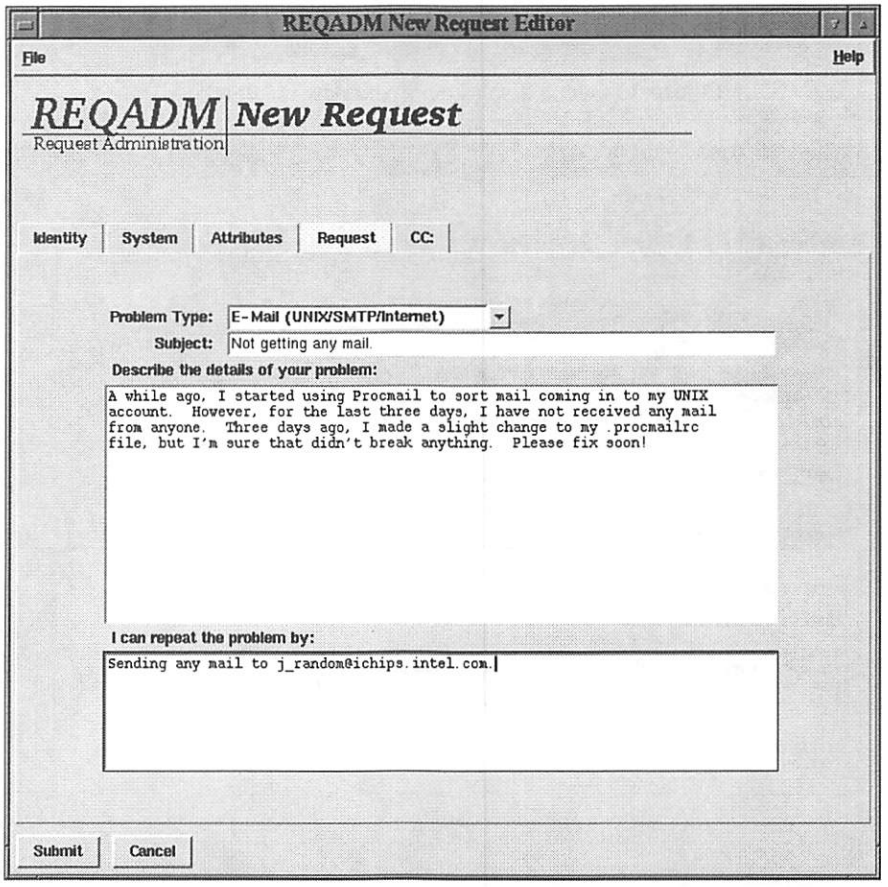


Figure 2: Request data-entry tool ("request" tab).

available UNIX engineer in the CSC, who will look at it, and hopefully be the one to actually own the ticket through to completion.

However, in some cases, the ticket must be passed up the line to an engineer who owns a specific area of the environment. For example, if the mailing list was beyond the capability of the CSC to handle (perhaps a bug in Majordomo, or a Sendmail anomaly), it would be re-assigned to "postmaster", which the system will automatically send to the current Postmaster-on-duty for the day. It could also have been assigned directly to a specific technician.

The person or group to whom the ticket is assigned is known as the "ticket owner" and is responsible for seeing that it gets resolved.

Here's where the different queueing models discussed above come into play. Since some rotations, like "postmaster", are staffed with specific people dedicated to own specific areas of the environment, the traditional "multiple queue, multiple server" model is an optimal way to supply waiting tickets to the specific area owners on duty.

For distribution of the bulk of our requests, however, which are handled by those working in the CSC, the newer "single queue" model is actually more effective. This blending of philosophies has proven quite successful for us.

Priority

Each ticket has a priority, which is on the same four-level scale as the "importance" rating set by the

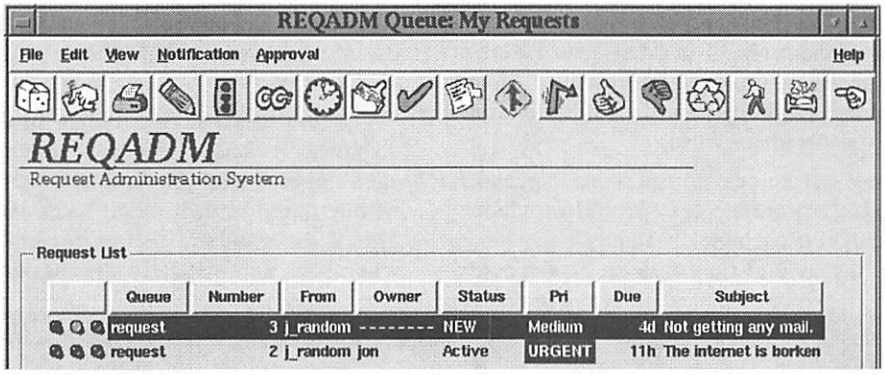


Figure 3: Queue display, showing new ticket arrival.

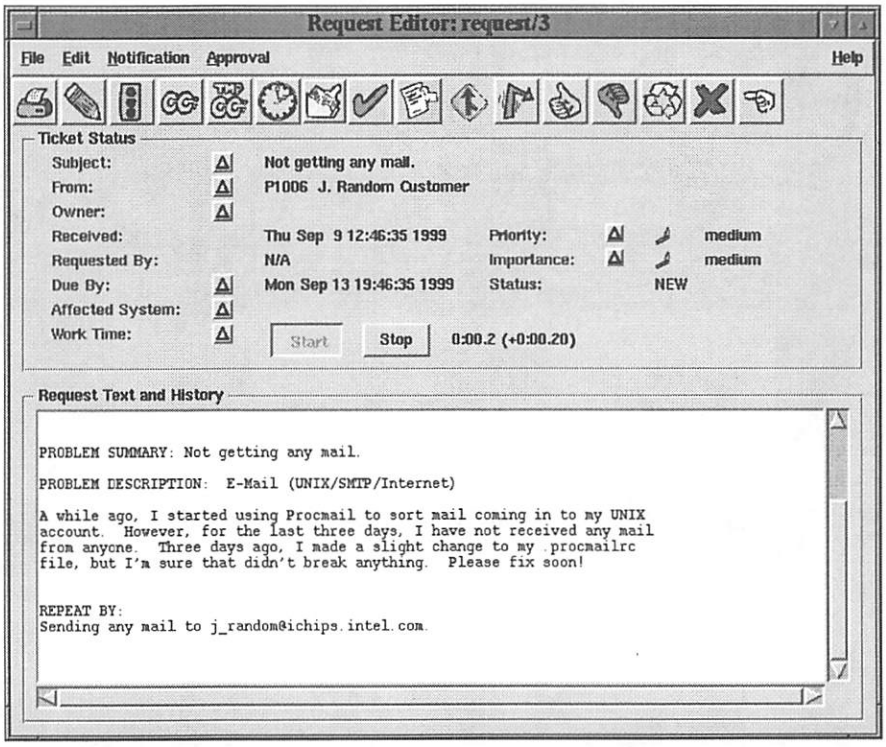


Figure 4: Examining a ticket.

customer. However, the importance rating is just an indicator from the customer. The true queue priority is set by the systems administrator. Generally, this will follow the customer's importance rating, unless EC disagrees with the customer as to the actual urgency of the problem.

When the systems administrator views her queue of tickets she owns, this list is sorted (by default) according to the amount of time remaining before it's due, which is a combination of the committed due date, priority and contractual SLA expectations.

Status Updates

As the ticket owner works on trying to solve the request, she will make updates, either as notes to herself and other support engineers, or to the customer, giving progress updates, asking questions, and recording what work was actually done.

These updates are made within the REQADM system, and are recorded in the permanent history of the request ticket. The results of each ticket editing session (which may include multiple actions such as updating the committed completion date, changing priority and adding a status note) is summarized by E-mail to the ticket submitter, owner, and other interested parties specified in the ticket's "CC:" list.

Checklists

Depending on the problem type being reported, the system may also attach pre-defined checklists of tasks which need to be followed to resolve that kind of issue. This assists the support engineer, especially if they are trying to perform an unfamiliar task during their CSC rotation week.

The items on the checklist must be completed before the ticket can be marked as resolved.

Holds

From the moment a request ticket enters our system, a "clock" starts ticking down the hours until it comes due according to our negotiated SLA with our customers. There will be times, however, when we are waiting for the customer to get back in touch with us before we can proceed any further.

In these cases, we place the ticket "on hold" which makes it clear to anyone scanning the queues that EC is not actively working on the ticket until some external event (usually communication from the customer) takes place. It also stops the "clock", effectively extending the due date by the length of time the ticket was on hold.

The next action logged to the ticket automatically removes the hold and returns it to active status.

Resolution

Finally, the customer's complaint is solved, and the support engineer is breathing a sigh of relief and feeling the satisfaction of a job well done. The ticket is marked as "resolved" and again the SLA clock stops counting against us (since we've completed our assigned task).

Categorization

At any point up to resolution, the ticket may be categorized. We have defined a multi-level menu hierarchy for specifying the root cause of the problems we see. By tagging each ticket with its root cause category, we can generate many interesting reports and analyze what parts of our environment might need extra attention.

For our example mailing list ticket, the engineer resolving the ticket would select the "mail" category, and the "mailing list problem" sub-category.

Supervisor Approval

In some rare cases, such as when the customer escalates their issue to EC management due to a disagreement or problem with the ticket owner, an EC manager must approve the resolution of the ticket. Normally, this is not necessary.

Customer Approval

This is where things get interesting. One of the problems we had with our old system was the perception among some customers that EC personnel were marking tickets as "closed" without fully solving the problem at hand. We had to keep track of which customers wanted us to never close a ticket without explicitly checking for their satisfaction first, and those who were annoyed if we bothered them to get that approval.

With the new system, we automated a customer approval process, which has made both sides happy. In this system, when EC thinks the ticket has been adequately completed, it's put into a new "resolved" state. The ticket is definitely no longer being worked on, but it's not really quite dead yet, either. It exists in a nether region or "ticket purgatory" for a while, and keeps its "resolved" status until approved by the customer.

At this point, the customer is sent e-mail informing them that the ticket has been resolved, and asking them to confirm this by approving the closure of the ticket. They may do so as simply as typing an X in the correct box and mailing the E-mail message back where it came from. Or, they may access the ticket using an interactive tool and click on the "approve" button.

Once this is done, the ticket becomes fully "closed" and is no longer a concern.

On the other hand, if the customer is still not satisfied, they may indicate that they deny the approval, and the ticket is automatically placed back to active status.

If the customer gets busy and doesn't bother explicitly approving closure of a ticket, the REQADM system will automatically close it after a grace period (currently five days) expires.

Resurrection

If a ticket has been fully closed for a while and the problem arises again, a customer or administrator

can search for it and reopen the ticket to resume active work once again.

Interfaces

One of the most important requirements of the request system is that it be easily accessed from just about any kind of environment.

Our customers were already used to submitting tickets via E-mail, so at a minimum that practice needed to remain possible. The challenge to that, of course, was to find a way to allow people to include ticket attributes such as importance and due date (q.v.), but still allow free-form text for everything else, and accept mail even if it doesn't contain special fields.

The solution we found was to have the mail receiver accept a "template" which provides special fields for all those attributes. If a random-text message is submitted without the template, a warning is sent back to the user advising them that they would get better results by including the template next time, allowing them to specify all the attributes appropriate to their issue.

Our preferred method of submitting tickets, however, is to use the GUI tool described above. For times when a windowing system is unavailable or undesirable, the *request* program will switch to ASCII mode and run the user through the ticket creation process, prompting for each line in series.

Likewise, ASCII versions of all the REQADM toolset are available when a GUI isn't an option.

To help facilitate automation of ticket submission and administration, all functions of REQADM may be performed using batch-mode commands, without requiring any user interaction at all.

There are also API libraries for writing Perl or C++ clients to the REQADM system, to accommodate any level of future interface desired.

To help remote users who wish to access the ticket system, or for those who simply prefer such things, there is also a fully-functional web interface for submitting and manipulating tickets in the system.

Personal Options

In a customer base as large as ours, there is no shortage of opinions for how the system should behave, what mail notices should look like, and every other aspect of the system.

Rather than try to compromise between everyone's desires, and recognizing that different people mentally organize data differently (and would benefit from different data presentation formats), we provide some personal configuration options. Every user has a "profile" maintained by REQADM which determines how they see mail formatted, whether to page them when tickets are assigned, and a large number of other factors.

E-Mail Correspondence

When anything happens to a ticket, mail is sent to the ticket owner, submitter, and everyone on the ticket's CC: list. Each person's copy is separately composed and mailed, so that the recipient's personal formatting preferences can be applied to their individual copy.

The mail body always begins with the comments placed by the person editing the ticket, since those will most likely be the most important text to consider. Following that is a summary of the actions taken and the current status of the ticket.

The mail headers are created in such a way that if anyone replies to the mail directly, it is sent back to REQADM itself. This reply is added to the ticket history and re-mailed to everyone involved in the request. In this way, we can exchange information with a customer or other party, and as far as the customer is concerned, it's just a simple E-mail conversation.

Nobody has to invoke a special tool to update the ticket along the way; REQADM just tracks the conversation automatically in the ticket history.

Technical Design

The REQADM system centers on a single server daemon (*reqadmd*). This server maintains its own database containing request tickets and personnel information.

Databases

The server contains its own lightweight embedded database engine[2] to rapidly organize and access request ticket information. The personnel database is fed from the following sources:

1. The UNIX password file, which provides a mapping of user ID number to username. This is helpful for REQADM clients to identify who is running them, and grant an appropriate level of access for each user, without requiring users to "log in" to REQADM with yet another password.
2. The "usertable" file. This is a simple ASCII file mapping username to employee ID number. REQADM actually uses employee ID numbers to identify users, since that needs to be a key which will never (or at least very rarely) change. Experience has shown that OS usernames and user ID numbers tend to change too often.
3. Corporate personnel file. This is a simple ASCII file which provides information about each person, including phone numbers, mailstop, office location, and employee ID number. This allows REQADM to help EC staff find customers when they need to follow up on a ticket.

Every night, REQADM checks these sources for any information that needs to be updated. (For

[2]It actually uses the Athena DBMS library from Software Alchemy, Inc.

example, if a person moved to a new office, the corporate personnel file will show their new location, and REQADM will pull that change into its copy of the database too.)

REQADM can also be configured to access some of this information on the fly to allow new people to use it even if they are not currently in its personnel database. Specifically, it'll try contacting:

1. A local LDAP server we have which lists all known E-mail addresses for all Intel employees (many people have accounts at multiple sites and may have mail addresses at each, as well as the overall "intel.com" domain). This will help REQADM resolve unknown (to REQADM) mail addresses to employee ID numbers, which it can look up further.
2. A local server providing the above-mentioned corporate personnel information. REQADM will use this to create personnel records in its database for new people when first encountered.

Whichever combinations of the above files and services reasonably apply to other sites' environments can easily be set up to support REQADM.

Additionally, the *reqadmd* server accesses the tickets stored in human-readable MH-format ASCII files (in pre-2.0 versions; after 2.0, REQADM stores all tickets in the database but writes out ASCII recovery files in quickly-parsable machine-friendly ASCII formats, Just In Case. Normally, though, those files are written but never read by REQADM 2.*x*). The server is the only process allowed to directly touch either the database or ASCII files; as such, it arbitrates security and locking amongst the request system users.

Clients

Everything else the sysadmins and customers encounter is a client program or script which contacts the *reqadmd* server via TCP/IP sockets, requests it to perform queries or modifications of tickets, and reports back to the user.

Clients exist for submitting tickets, setting personal look-and-feel preferences, browsing the queues and editing request information.

Most of the clients respond to whatever environment they sense they are being run from (GUI, ASCII, WWW, or batch) and interact with the user appropriately.

Supporting Scripts

A number of scripts support the operation of REQADM. The most interesting of them are listed below:

REQMGR manages automatic assignment of tickets to sysadmins who take turns being "on duty" for specific tasks such as postmaster, webmaster, and account creation. It checks schedule files to see who the candidates are to receive each ticket, and then checks the REQADM server to see who has the smallest active request load.

Reqadm-mail-receiver takes all the incoming E-mail (whether new incoming tickets or mailed responses to existing tickets) and invokes the other clients necessary to get the mailed information into REQADM.

Reqadm-auto-close runs as part of the nightly maintenance procedures, finding all tickets which have been marked as "resolved" but have not been touched for a designated number of days. These are considered to be abandoned by the customer, who has not responded to approve or deny the resolution. Those tickets are then marked "approved".

Extension APIs

In order to allow metrics-gathering scripts and other new clients to be written as desired, a comprehensive C++ API library is provided allowing virtually any new facility to be added to the system.

A reasonable subset of those functions are available in a Perl module, allowing scripts to access the REQADM server. We make significant use of this module for our statistical analysis scripts.

Implementation Notes

The core of REQADM (server, clients, libraries) consists of nearly 42,000 lines of code (mostly C++, with a bit of Tcl/Tk and C as necessary). This is supplemented by about a dozen shell and Perl scripts responsible for the ongoing maintenance tasks.

It has been built successfully on Solaris, AIX, HP/UX, SunOS, Linux, FreeBSD and Windows NT. (Currently the server must be SPARC Solaris or FreeBSD/i386.)

Phase I: Gradual Integration

The REQADM program took almost a year to go from initial design committee meetings to the completed version 1.0 release. At that point (early 1998), we were ready to cut over from the old MH system to the new REQADM server.

We needed to be extremely cautious, however, since the request ticket system is so critical to our department. In case of trouble, we needed a fast escape route to switch the old system back online.

The major consideration to that end was the decision to keep the old MH-style file format from the old system. REQADM was designed to read and write the old-style files, and maintain an "overview" database tracking the status and ownership of the tickets in the discrete disk files. This allowed for fast database queries and reports of this overview data, but if the actual contents of a ticket needed to be examined, the disk file had to be opened and its text format parsed.

In this way, if REQADM failed in some major way, we could simply turn off the REQADM server and continue using our old scripts on the same data files, with no loss of information or stoppage of work.

We began with an "alpha test" phase internally within the EC group. We started a new REQADM server without an existing set of tickets, and ran

scripts to randomly generate a few hundred request tickets. These tickets contained instructions for the owner to perform 6-8 randomly chosen tasks within the request system, such as "Assign this ticket to *(random other person)*", or "Change the priority to 'urgent'". We used these tickets to further test the REQADM system, as well as to train our support staff on the new interface they'd be using.

Once we were satisfied with that, we cleared out the random tickets and asked our support staff to use REQADM for all internally-generated tickets. We also asked a select group of customers to try using REQADM for all their new issues as well. This gave us more "real world" testing of REQADM, while allowing us to still use the old system with most of our existing customers.

To make this work, we hacked REQADM to start issuing new request tickets at ticket number 70,000, which was a significant gap from the largest ticket number in the MH system (which was still under 60,000 at the time).

When we believed REQADM was ready for production use, we ran a Perl script to import all the MH-style tickets into the REQADM database (adding them to the higher-numbered ones which were already there from the customer testing period), and told the customers that REQADM was the way to submit tickets from then on.

We also provided them with a step-by-step tutorial for each of the customer interfaces to REQADM. This tutorial allows them to submit sample tickets to an "autotest" queue, which is ignored by humans, but monitored by a cron script which will pretend to be a sysadmin working on their ticket, doing random things to it and eventually closing the ticket. This lets them figure out how the system works without unduly annoying any of the EC staff.

REQADM was a success and the emergency back-out plan never had to be executed.

Phase II: Taking the Plunge

As this paper is being written, we are beginning implementation of REQADM version 2.0, which will include quite a number of improvements to the system described here. By the time you read this paper, 2.0 should be completed and ready for release.

One of the major changes will be to abandon the old MH format, and to store all ticket information completely inside REQADM's database. This will allow us to search for tickets by text content as well as attribute and subject information, and will improve the speed of handling tickets.

Once we do this, we won't have the "safety net" of falling back to the MH scripts, but since using REQADM for over a year and a half, we don't believe that will be an issue any longer. The 2.0 system will have its own internal recovery systems to help prevent loss of data in the event of database corruption or other problem.

Success of REQADM

Our customers and support staff have been very happy about the improvements REQADM has provided to our ticket handling process in the PMD EC. We've been gratified to have been asked by other Intel groups to give them copies of REQADM to use for their own ticket management.

According to programming folklore, one of the marks of success for a software system is when someone successfully applies it to a problem which the programmer never anticipated. This happened for REQADM when one of our product testing groups decided to adopt REQADM to track downtimes for their chip- and board-tester systems, and for keeping maintenance information with the vendors who supply those systems. By keeping the REQADM tickets' timestamps updated accurately, and by asking for just the right metrics from the REQADM server, they now are able to get automatic mean-time-between-failure and mean-time-to-fix statistics for their test equipment, in addition to the normal usage of REQADM to track active issues with their staff and vendors.

Fringe Benefits

The EC group has also been able to leverage REQADM to do more for us than just track requests for help from our customers.

For example, we created a queue of requests the EC group places with outside vendors. Now when we call (for example) Auspex to get a new fileserver part, we can create a ticket and assign it to our Auspex sales or service representative. The vendor can use the mail interface to REQADM to add updates and other information to the request ticket along the way, giving us a permanent record of service for our systems. When we pull a service ticket for viewing in REQADM, we can ask for all other tickets filed against the same system, and see trends that might need to be discussed with the vendor.

Metrics Gathering

One of the most important topics for EC managers to consider is how to keep track of the general issues the EC group is facing, and how to reduce the number of backlogged tickets (by getting our teams to work more efficiently), and to reduce the number of tickets coming *in* to the system (on the theory that if we proactively improve the quality of system services, the customers won't have so much to complain about).

In order to do this, they need real data to work from. We've been gathering various statistics on weekly numbers of request tickets opened, closed, backlogged, late, and so forth, which get translated to various graphs and charts for managers to look at and discuss in their meetings.

The system also tracks the "categories" assigned to each ticket by the ticket owner. These categories show the root cause of the problem which prompted the ticket to have been entered in the first place.

The categories are organized into multiple tiers. At the outermost level we might have a list of categories like:

- Account Administration
- Fileserver/Disk Management
- E-Mail
- Network/Internet
- Performance Issues
- Printing/Plotting
- User Environment

Under each of those are sub-categories and sub-sub-categories. For example, under "E-Mail" we might find:

- Bounced, Pilot Error
- MTA Problem
- MUA Problem
- Bogus .forward File
- Spam/Abusive Mail

We generate reports at the top level each week ("How many mail requests did we get? How many network problems?"). We also developed an interactive Request Analysis Tool ("RAT") on the web which allows one to browse these statistics, opening up a top-level category to see *why* there were so many tickets of a particular type. From these statistics, we can recognize problems in the environment early enough to reduce their impact.

Multi-Site Use

Our support group is divided between geographical locations, with some of us in Oregon and others in Washington. While, for the most part, the Oregon and Washington teams don't tend to work together on individual tickets, there is still a need to have our request systems accessible to each other.

We have had reasonable success having those at remote sites run clients which connect to our server, giving them access to our request tickets. However, this doesn't work well when the wide-area network between Oregon and Washington goes down.

We're developing a new protocol which will allow two or more REQADM servers to communicate between themselves, passing information occasionally between them. This will allow our Washington team to maintain their own server and local request queues. When they get a ticket that belongs to the Oregon team, they can just click on a "cross-site transfer" button and the server will move the ticket to the Oregon server. This feature will also allow people at either site to browse (but not necessarily modify) queues at remote sites.

Future Development

There is always room to improve any system, and we'll probably never be 100% finished with REQADM, as long as people keep asking for new features.

One thing we've considered is a tie-in to some kind of expert system or knowledge base. We'll probably do this in two places. First, when the customer is submitting a ticket, it would be helpful to have the system refer them to information and/or URLs offering advice for the kind of problem being submitted. This may even help reduce the number of incoming tickets.

Second, on the support side, we would like to expand REQADM's capability to cross-reference past tickets to see what was done previously to a ticket, as well as accessing a knowledge base of information about all areas of our environment.

We're also working on setting alarms for certain kinds of events (e.g., to warn when a ticket is about to become due), setting time-outs for the "on hold" feature, etc.

The idea of replicated backup servers for REQADM is also appealing, and we may pursue that at some future point.

Glossary

CSC Computing Support Center. The physical location where front-line and second-tier personnel sit during their week-long rotation. Phone support is handled there, as well as walk-in traffic.

EC Engineering Computing. The overall team of systems administration staff for our engineering department of customers. Contrast with IT.

IT The corporate Information Technology group. The computing support organization for everyone else in the company not served by Engineering Computing.

MCC The Monitoring and Control Center. The old name for the CSC, before someone thought it sounded too Orwellian.

POD Affectionate term for the CSC; see [4].

queue A collection of request tickets. The separate queues help organize tickets into broad classifications.

rotation A "group" which may be assigned a ticket. Through some manual or automatic means, a ticket is then dispatched out of that rotation to the next available real person who is serving a duty rotation for that kind of issue.

SLA The Service Level Agreement between EC and our customers. This specifies the expected timeframes wherein tickets should

be resolved, and what services we are willing to support, in what manner.

References

1. David H. Crocker, "Standard for the Format of ARPA Internet Text Messages," *RFC 822*, 1982.
2. Rémy Evard, "Managing the Ever-Growing To-Do List," *Proceedings of the Eighth USENIX Systems Administration Conference (LISA VIII)*, USENIX, San Diego, CA, 1994.
3. Tinsley Galyean, Trent Hein, and Evi Nemeth, "Trouble-MH: A Work-Queue Management Package for a >3 Ring Circus," *Proceedings of the Fourth USENIX Large Installation Systems Administration Workshop (LISA IV)*, USENIX, 1990.
4. William Goldman, *The Princess Bride, 25th Anniversary Edition,* Ballatine, 1998.
5. Help Desk Institute, *http://www.HelpDeskInst.com.*
6. D. Johnson, "NOC Internal Integrated Trouble Ticket System; Functional Specification Wishlist," *RFC 1297*, 1992.
7. Bryan McDonald, "QMH: A Problem Tracking System," *Proceedings of the World Conference on System Administration and Security*, 1992.
8. Joe Rhett, "Request v3: A Modular, Extensible Task Tracking Tool," *Proceedings of the Twelfth Systems Administration Conference (LISA '98)*, p. 327, USENIX, Boston, MA, 1998.
9. Elizabeth D. Zwicky, "Getting More Work Out Of Work Tracking Systems," *Proceedings of the Eighth USENIX Systems Administration Conference (LISA VIII)*, USENIX, San Diego, CA, 1994.

Author Information

Steve Willoughby is a Senior Systems Programmer at Intel's Performance Microprocessor Division, where he has been for the last six years playing (at various times) Auspex Administrator, Postmaster, Security Czar, Perl Programming Teacher and maker of internal software applications.

He discovered Version 7 UNIX while in high school (ca. 1980) and, apart from brief forays into VMS in college and failed attempts to hide from a couple of other operating systems, he's been spending most waking hours since then tinkering on UNIX in one form or another, either writing software or administering systems.

He lives in the Portland, Oregon area with his wife, son and assorted small furry creatures. He keeps a vintage Altair 8800 as a pet. In his spare time, he tries to avoid running a MUD system (*http://www.rag.com*). He can be reached by E-mail at <steve@ichips.intel.com>.

APPENDIX

This appendix shows a comparison between REQADM and other popular free ticket tracking systems.

This information was taken from the documentation accompanying the following packages. Not all features were clearly and thoroughly documented in each case, so this table may not be 100% complete. Time did not allow for us to install and extensively use each package for an in-depth comparison.

REQADM Version 1.2, with anticipated features which should be completed when this paper goes to press.

RFC 1297 The mythical ticket tracking system described in the RFC is compared with the real features of the existing tools.

Gnats Version 3.2 from ftp://prep.mit.edu/pub/gnu/gnats/gnats-3.2.tar.gz

Netlog Version 2.4 from ftp://ftp.jvnc.net/pub/netlog-tt-2.4.tar.Z

PTS Version 1.1a2 from ftp://ftp.x.org/contrib/pts-1.1a2.tar.gz and http://www.halcyon.com/dean/pts/pts.html

Req Version 1.2.7 and tkreq version 1.10 from ftp://ftp.ccs.neu.edu/pub/sysadmin/*

RUST Version 1.0b6pl2 from ftp://ftp.eng.utah.edu/pub/sys-admin/rust/rust-1.0b6pl2.tar.gz

Request Version 3 from http://www.noc.isite.net/software/Request/ was not compared since the web site did not contain an active link to obtain a copy of the software, and did not have documentation on line to describe all features in depth. However, from what information we have, it appears to be comparable to the other systems shown.

Feature	REQADM	RFC 1297	Gnats	Netlog	PTS	Req	RUST
General Features							
Running history in ticket	yes	yes	yes	yes	yes	yes	yes
Auto-timeout for escalation	no	yes	no	no	yes	no	no
Feedback to net monitors	*	yes	no	no	no?	no?	no
Fixed-input field templates	yes	yes	yes	no	yes	no?	no
Free-form text fields	yes	yes	yes	yes	yes	yes	yes
Problem classes or queues	yes	yes	yes	no	yes	no	yes
Assisted entry for fields	yes	yes	yes	yes	yes	no?	no
Help screens in tools	yes	yes	no?	no	?	no?	no?
Pass tickets to other site	yes	yes	no	no	no	no	no
Integrated with expert sys	no	yes	no	no	no	no	no
API for ad hoc queries/rpts	yes	yes	no	no	yes	no	?
API for custom clients	yes	yes	no	no	yes	no	?
Refer to old closed tickets	yes	yes	no?	no	yes	yes?	yes?
Re-open a closed ticket	yes	no	no?	no	yes	yes	yes
Database used	custom	N/A	no	no	custom	no	no
Merge duplicate tickets	yes	no	no	no	no	yes	yes
Put ticket "on hold"	yes	yes	no	no	no	no	no
Support Rotations	yes	no	no	no	no	no	no
Auto-assign to Rotations	yes	no	no	no	no	no	no
Queue Displays							
Auto-updating ticket disp.	yes	yes	yes	no	no?	no	no
Select view by attributes	yes	yes	yes	no	no?	yes	yes
Sortable by owner/submitter	yes	yes	no?	no	no?	no	no
Sortable by ticket priority	yes	yes	no?	no	no?	no	no
Sortable by queue/number	yes	yes	no?	no	no?	no	no
Shows ticket status/age	yes	yes	yes	no	yes	yes	yes
Customizable displays	yes	no	no	no	no	no?	yes
Phonetic subject searches	yes	no	no	no	no	no	no
Security							
Admin functions restricted	yes	yes	yes	yes	yes	no?	yes
Per-field ACL/permissions	no	yes	no	no	no	no	no
Admin-eyes-only notes	yes	yes	no?	no	no	no	no
Customers can note tickets	any	*	no	no	no	any	any
Customer approval for close	yes	no	no	no	no	**	**
Admin approval for reopen	no	no	no	no	yes	no	no

Feature	REQADM	RFC 1297	Gnats	Netlog	PTS	Req	RUST
Ticket Attributes							
Customer priority levels	4	yes	3	no	no	3	3
Support priority levels	4	yes	3	no	no	no	no
Time/date submitted	yes	yes	yes	yes	yes	yes	yes
Time/date resolved	yes	yes	yes	yes	yes	yes	yes
Time/date last updated	yes	yes	no?	no?	no?	yes	yes
Location/system of user	yes	yes	no	no	no	no	no
Machine/network involved	yes	yes	no	no	no	no	no
One-line problem summary	yes	yes	yes	yes	yes	yes	yes
Next action to be taken	yes	yes	yes	yes	yes	yes	yes
Ticket owner (comp support)	yes	yes	yes	no	no?	yes	yes
Time spent working on prob	yes	yes	no	no	no	no	yes
Time ticket took to close	yes	yes	yes	yes	yes	yes	yes
Supervisor approval of work	no	yes	no	no	no	no	no
Escalation to non-EC or mgt	no	yes	no	no	no	no	no
SLA due-date for prob type	yes	yes	no	no	no	no	no
SLA due-date for priority	yes	yes	no	no	no	no	no
Ticket status levels	8	yes	5	2	3	4	3+
Committed (manual) due date	yes	no	no	no	no	yes	yes
Requested (manual) due date	yes	no	no	no	no	no	no
Reports							
Summarize by network/host	yes	yes	no	no	no	no	no
Summarize by root cause	yes	yes	no	no	?	no	no
Summarize by date	yes	yes	no	yes	?	yes	yes
Mean Time Between Failure	yes	yes	no	no	*	no	?
Mean Time to Repair (work)	yes	yes	no	no	no	no	?
Graphical chart output	yes	yes	no	no	*	no	no
Environment							
Runs w/native window system	yes	yes	yes	yes	yes	yes	yes
Mult. open ticket windows	yes	yes	yes	yes	?	no	no
Auto. tools open tickets	yes	yes	yes	yes	yes	yes	yes
Pulls employee info from db	yes	yes	no	no	no	no	no
Pulls host/net info from db	yes	yes	no	no	no	no	no
Query env. when ticket open	yes	yes	*	no	*	*	*
E-mail notification/updates	yes	yes	yes	no	yes	yes	yes
Mail traffic logged to tkt	yes	yes	no?	no	no?	yes	yes
Batch updates w/o using GUI	yes	yes	yes	no	no?	*	yes
Failover to backup CPU/db	no	yes	no	no	no	no	no
X GUI interface available	all	N/A	no?	no	all	all	all
NT GUI interface available	all	N/A	no?	no	no	no	no
Web interface available	all	N/A	no	no	all	no	no
E-mail interface available	sub	N/A	sub	no	no?	sub	sub
ASCII interface available	all	N/A	all	no	all	all	all
Curses interface available	no	N/A	no	yes	no	no	no
Batch cmd interface avail.	all	N/A	all	no	no	all	all
EMACS interface available	no	N/A	yes	no	no	yes	no?
Free-form text submissions	yes	N/A	no	yes	yes	yes	yes
Assisted submissions	yes	N/A	yes	yes	yes	no?	no?
Users' custom preferences	yes	no	no	no	no	no	no
Mailed new-ticket receipt	yes	yes	yes	no	no?	yes	yes

* Possible to add by writing a custom script in Perl or other language.

** Possible by using existing features of the tool, but those features may not have been designed specifically (or exclusively) for that purpose.

all All major features are available in this mode.

sub A subset of features are available in this mode (e.g., you can submit new tickets by mail, and can add commentary to them, but can't otherwise administrate them via mail alone).

GTrace – A Graphical Traceroute Tool

Ram Periakaruppan and Evi Nemeth – University of Colorado at Boulder and Cooperative Association for Internet Data Analysis

ABSTRACT

Traceroute [Jacobson88], originally written by Van Jacobson in 1988, has become a classic tool for determining the routes that packets take from a source host to a destination host. It does not provide any information regarding the physical location of each node along the route, which makes it difficult to effectively identify geographically circuitous unicast routing. Indeed, there are examples of paths between hosts just a few miles apart that cross the entire United States and back, phenomena not immediately evident from the textual output of traceroute. While such path information may not be of much interest to many end users, it can provide valuable insight to system administrators, network engineers, operators and analysts. We present a tool that depicts geographically the IP path information that traceroute provides, drawing the nodes on a world map according to their latitude/longitude coordinates.

Introduction

Today's Internet has evolved into a large and complex aggregation of network hardware scattered across the globe, with resources accessed transparently with respect to their location, be it in the next room or on another continent. As the Internet becomes increasingly commercialized among many different corporate administrative entities, it is more difficult to ascertain the geographical routes that packets actually travel across the network. Knowledge of these geographical paths can provide useful insight to system administrators, network engineers, operators and analysts.

It is challenging to obtain the location for a given node of a path since there is no existing database that accurately maps hostnames or IP addresses to physical locations. Although RFC 1876 [RFC1876] defined a DNS resource record to carry such location information (the LOC record) for hosts, networks and subnets, very few sites maintain LOC records. Hence there is no straightforward way to determine the physical location of hosts.

GTrace is a graphical front end to traceroute that uses a number of heuristics to determine the location of a node. Often the name of a node in the path contains geographical information such as a city name/ abbreviation [Wood98] or airport code [Halsey98]. GTrace operates on the assumption that these codes and names indicate the physical location of the node. The locations obtained are connected together on a world map to show the geographical path that packets take from the source to destination host. GTrace also tries to verify the validity of each location obtained, eliminating ones that are incorrect.

The following sections review the traceroute tool and describe the design and implementation of GTrace. We also show example output from GTrace.

Traceroute

Traceroute is a tool that discovers the route an IP datagram takes through the Internet from a source host to a destination host. It works by exploiting the TTL (Time To Live) field of the IP Header. Each router that handles an IP datagram decrements the TTL field. When the TTL reaches zero, a router must discard the packet and send an error message to the originator of the datagram.

Traceroute uses this feature, initially sending a datagram with the TTL set to one. The first router along the path, upon receiving the datagram decrements the TTL, discards the datagram and sends back an ICMP error message. Traceroute records this first IP address (source address of the error message packet) and then sends the next datagram with the TTL set to two. This process continues until the datagram finally reaches the target host, or until the maximum TTL threshold is reached.

Design and Implementation of GTrace

Recognizing that it is not possible to obtain precise physical location information for all existing IP addresses, our main design criteria for GTrace was that it be sufficiently flexible to support the addition of new databases and heuristics. We chose to implement GTrace in Java, for both its portability and its new Swing [Swing] user interface toolkit. GTrace operates in two phases. In the first phase GTrace executes traceroute to the destination host and tries to determine locations for each node along the path. During the second phase, GTrace verifies whether the locations obtained in the previous phase are reasonably correct.

GTrace is composed of the following seven key components: Graphical User Interface, Dispatcher Thread, Hop Threads, Lookup Client, NetGeo Server, Lookup Server and Location Verifier. Figure 1 illustrates the overall architecture of the tool. The function of each component is described below.

Graphical User Interface

The Main Thread handles all features of the Graphical User Interface and is responsible for

spawning the dispatcher thread when a destination host is specified. Figure 2 shows a snapshot of GTrace on startup. The GUI has two sections, with a map

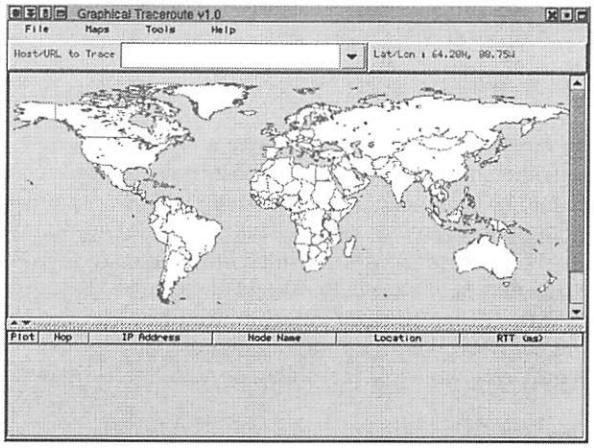


Figure 2: GTrace's startup screen.

on the top and traditional traceroute output below. The tool supports zooming in or out of particular regions of the maps. Twenty-three maps are available courtesy of VisualRoute [VisualRoute] and users can also add their own. We later provide an example that highlights some of the features of the GUI.

Dispatcher Thread

The function of the dispatcher thread is to execute traceroute to the destination host. It then reads the output of traceroute, creating a new thread for each line of output. These threads are referred to as hop threads. The dispatcher thread can also read traceroute output from a file, which allows users to visualize traceroutes performed using third-party traceroute servers.

Hop Threads

Each hop thread parses its line of traceroute output and immediately notifies the main thread so that it can update the display with relevant traceroute fields for the corresponding hop. It then creates an instance of the Lookup Client, which tries to determine the location of the node and return the resulting information to the main thread before exiting.

Lookup Client

The Lookup Client tries to determine the location of a node by using a set of search heuristics. Many of the nodes in a typical traceroute path are in the '.net' domain. Often the names of these nodes have some

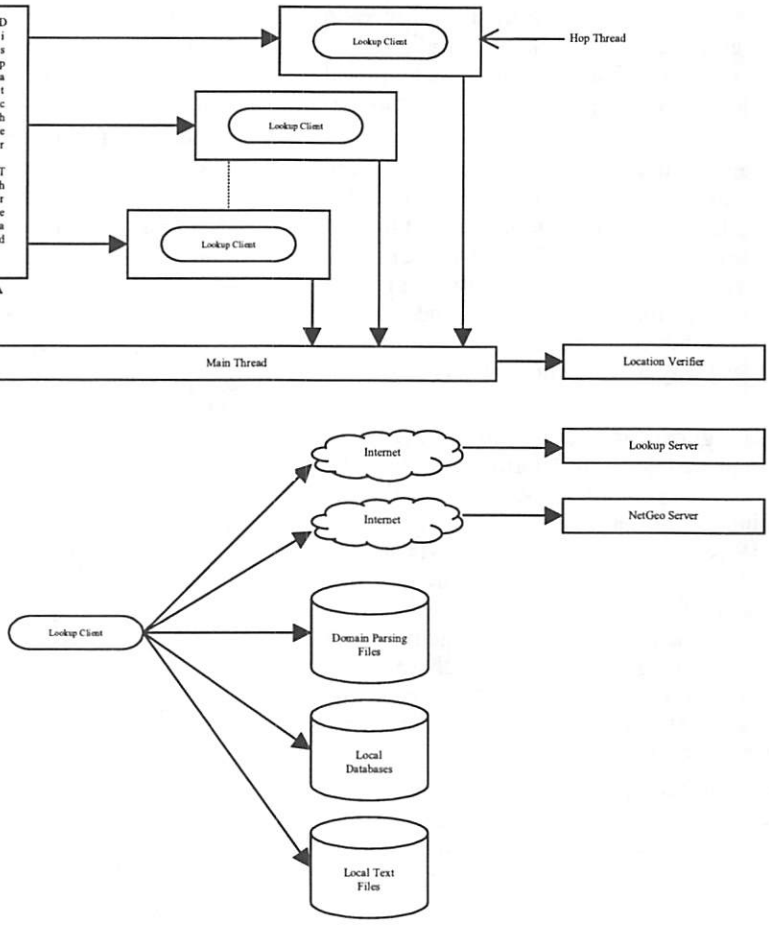


Figure 1: GTrace architecture.

geographical hint in them. The Lookup Client uses customized domain parsing files that specify rules for extracting these geographic hints. We have such files for several '.net' domains that use internally consistent naming conventions within their domain.

However this technique does not solve the problem of locating nodes that do not have embedded geographical hints. GTrace also utilizes databases from CAIDA [DBCAIDA] and NDG Software [DBNDG] that map hostnames and IP addresses to latitude/longitude coordinates. For nodes with no information in these databases, the Lookup Client uses the domain's registered address (unfortunately often only the headquarters for a geographically distributed infrastructure) obtained through a whois lookup to determine the location. Nodes for which the Lookup Client is unable to determine a location are listed in the text portion, but skipped in the geographical display.

The search algorithm is described below. We try each heuristic in turn, stopping as soon as one yields a location. The Lookup Client also makes a note of the search step that produced the location, providing this information to the user as well as the Location Verifier.

Search Algorithm

1. Check the cache to see if the location for the IP address has already been determined from a previous trace.
2. Check if the host has a DNS LOC record. If not, reduce the hostname to the next higher level domain (i.e., remove the first component of the name) and check again for a LOC record. Continue until we have reached the last meaningful component of the name (for example foo.com in xxx.foo.com or bar.com.au in xxx.yyy.bar.com.au). Note that if a site has a LOC record for the whole domain, but machines are located outside the scope of that LOC record, GTrace would end up using incorrect data. If the Location Verifier detects such a situation, GTrace will notify the user and optionally can be configured to notify GTrace's author, who will contact the DNS administrator at the corresponding site to correct their LOC records.
3. Search for a complete match of the hostname/IP address in the databases and files specified in the GTrace configuration file.
4. If the hostname has a corresponding domain parsing file, use the rules defined in the file to extract geographical hints and proceed as indicated in the file.
5. Reduce the hostname to the next higher level domain as in step two and search for a match as in step three. The process is repeated until we have reached the last meaningful component of the name.
6. Query the NetGeo [NetGeo] server with the IP address. NetGeo determines the location based on whois registrant information.
7. If still no match occurs and the last two letters of the hostname end in a two-letter country code, map it to the geographic center of that country.

The search algorithm is ordered in decreasing level of location reliability. Locations obtained from steps 2 and 3 are taken as authoritative, while those from step 4 onward are considered a guess. Cache entries will indicate whether the location was authoritatively determined or was a guess; this status determines the color of the lines connecting the nodes on the map.

The Lookup Client does not determine locations for IP addresses that fall in the ranges 10.0.0.0-10.255.255.255, 172.16.0.0-172.31.255.255 or 192.168.0.0-192.168.255.255, as these blocks are reserved for private internet use [RFC 1918]. Unfortunately some addresses in these blocks do occur in traces since some ISPs use this address space for internal router interfaces. These nodes are shown in the text portion of the display with the location marked as private internet use.

The Lookup Client queries the Lookup Server if one is defined in the GTrace configuration file and if location information has not been obtained through step 1, 2 or 3 of the search algorithm. GTrace compares the reply from the Lookup Server with any obtained previously from local lookups, with preference given to the location obtained through a lower numbered search step. Based on the GTrace configuration file, the Lookup Client also uses databases, text files and domain parsing files as follows.

Databases

The Lookup Client may need to perform lookups in many databases before determining a location. GTrace's database support is provided by the BerkeleyDB [BerkeleyDB] embedded database system, which supports a Java API that the Lookup Client uses to query the databases. The database interface allows multiple thread reads on the same database at the same time. Locking is not an issue, since Lookup Clients only read, do not write.

Five databases are packaged with the GTrace distribution:

Machine.db [DBCAIDA]	Maps machine names to their latitude/longitude values.
Organization.db [DBCAIDA]	Maps organizations to their latitude/longitude values.
Hosts.db [DBNDG]	Maps IP addresses to their latitude/longitude values.
Cities.db [DBCAIDA]	Maps cities around the world to their latitude/longitude values.
Airport.db [AirportCodes]	Maps airport codes to their latitude/longitude values.

One can add a new database in BerkeleyDB format to GTrace with GTraceCreateDB and by adding an entry to the GTrace configuration file. The contents of the database, i.e., whether it maps hostnames, IP addresses, or both to latitude/longitude values, also have to be indicated in the configuration file. The user can also add records to existing databases using GTraceAddRec. GTraceCreateDB and GTraceAddRec are Java classes packaged with the GTrace distribution.

Text Files

Users may also specify new locations for nodes in text files, though it is more efficient to create a database for large data sets. New files have to be listed in the GTrace configuration file in order for the search algorithm to have access to them.

Domain Parsing Files

Files describing properties of each domain are used to ferret out geographical hints embedded in hostnames. These files define parsing rules using Perl5 compatible regular expressions. GTrace uses the regular expression library from ORO Inc. [ORO-Matcher] for parsing. New files can be added and existing ones modified without requiring any changes to GTrace.

For example, ALTER.NET (a domain name used by UUNET, a part of MCI/WorldCom) names some of their router interfaces with three letter airport codes as shown below:

```
193.ATM8-0-0.GW2.EWR1.ALTER.NET
(EWR -> Newark, NJ)

190.ATM8-0-0.GW3.BOS1.ALTER.NET
(BOS -> Boston, MA)

198.ATM6-0.XR2.SCL1.ALTER.NET
(Exception)

199.ATM6-0.XR1.ATL1.ALTER.NET
(ATL -> Atlanta, GA)
```

Figure 3 shows an example of a GTrace domain parsing file that would work for ALTER.NET hosts. The file first defines the regular expressions, followed by any domain specific exceptions. The exceptions are strings that match the result of the regular expressions. The user may identify the exception's location either by city or by latitude/longitude value using the format shown below:

```
exception=city,state,country
          city,country
          L: latitude, longitude
```

In the former case, the user should also use GTrace-QueryDB to ensure that the cities database has a latitude/longitude entry for the city specified. The first line in Figure 3 defines a substitution operation, which when matched against 193.ATM8-0-0.GW2.EWR1.ALTER.NET, would return 'EWR'. The contents following the last '/' of the first line indicate what to do with a successful match, namely in this case to instruct the program to first check for a match in the data specified in the current file and then for a match in the airport database.

The reason for checking the domain parsing file first is that sometimes the naming scheme for a given domain is not consistent. For example, a search for SCL obtained from 198.ATM6-0.XR2.SCL1.ALTER.NET in the airport database would return a location for Santiago de Chile. In the case of ALTER.NET, they also use three letter codes that are not airport codes but abbreviations for US cities (Figure 3 illustrates three such abbreviations). Note that if this exception list were not present and SCL did get mapped to Chile, the Location Verifier would likely have eliminated it using the Round Trip Time (RTT) heuristic described later, which would have recognized the RTT as much too small to get a packet to Chile and back.

Sometimes ISPs name their hosts with more than one geographical hint in them. For example VERIO.NET names some of their hosts in the following format: den0.sjc0.verio.net, which typically suggests source and destination of the interface. If there is no rule on whether the convention is to use the source or destination label first in the hostname, the rule could be defined to extract both and GTrace could use the Location Verifier's heuristics to guess.

The advantage of this technique is that one can describe an entire domain as a set of rules without needing database entries for every host in the domain. The limitation of the technique is that it will fail for domains that do not use internally consistent naming schemes.

NetGeo Server

The original design of the Lookup Client performed and parsed results of whois lookups directly, which required storage of a prohibitively large number of mappings of world locations to latitude/longitude values. Distributing such a large database with GTrace was not ideal. CAIDA's NetGeo [NetGeo] tool, with its ability to determine geographical locations based on the data available in whois records, provided a vital resource.

```
s/.*?\.([^\.]+)\d\.ALTER\.NET/$1/this,airport.db
scl=santaclara, ca, us
tco=tysonscorner, va, us
nol=neworleans, la, us
```

Figure 3: Example of a domain parsing file for ALTER.NET.

NetGeo is a database and collection of Perl scripts used to map IP addresses to geographical locations. Given an IP address, NetGeo will first search its own local database. If a record for the target address is found in the database, NetGeo will return the requested location information, e.g., latitude and longitude. If NetGeo finds no matching record in its database, it will perform one or more whois lookups until it finds a whois record for the appropriate network. The NetGeo Perl scripts will then parse the whois record and extract location information, which NetGeo both returns to the client and stores in its local database for future use.

The NetGeo database contains tables for mapping world location names (city, state/province/district, country) or US zip codes to latitude/longitude values. Most whois records provide enough address information for NetGeo to be able to associate some latitude/longitude value with the IP address. Occasionally the whois record only suggests a country or state, in which case NetGeo returns a generic latitude/longitude for that country or state. In preliminary testing, NetGeo has been able to parse addresses and find (albeit sometimes imprecise) latitude/longitude information for 89% of 17,000 RIPE whois records, 76% of 700 APNIC whois records and for more than 95% of 30,000 ARIN whois records.

Lookup Server

The Lookup Server handles requests from Lookup Clients and tries to determine the location of a host or IP address by executing steps three, four and five of the search algorithm. This information is sent back to the client, which then decides whether to use the location information or not depending on the locations it might have received from other Lookup Servers or lookups it performed locally. The Lookup Client selects the location that was obtained from the lowest numbered search step. The Lookup Server can also be requested by the Lookup Client to execute step two of the search algorithm. This is because not all versions of nslookup support queries for LOC records. GTrace tests the version of nslookup on the machine it is running on to determine if such a request is necessary.

Location Verifier

The Main Thread invokes the Location Verifier once all the hop threads have died and the trace is complete. The task of the Location Verifier is to check whether the locations obtained for nodes along the path are reasonable. The verifier does not determine new locations for nodes, it only indicates to the user why an existing location might be wrong and where the node could possibly be located.

The verifier algorithm is based on the fact that IP packets can not travel faster than the speed of light. Light travels across different mediums at different speeds: 3.0×10^8 m/s in vacuum, 2.3×10^8 m/s in copper and 2.0×10^8 m/s in fiber [Peterson]. GTrace uses the speed of light in copper for all of its calculations.

For each successive pair of hops that have locations, the verifier algorithm uses the deltas of the round-trip times (RTT) returned by traceroute to rule out locations that are physically not possible. Traceroute measures RTT rather than one way latency, as this would require control over both end nodes and delays are often not symmetric. Also, one must be cautious

Hop	Node Name	IP Address	Search Step	Location	RTT (ms)
1	pinot-fe2-0-0	(192.172.226.65)	6	San Diego	0.917 ms
2	medusa.sdsc.edu	(198.17.46.10)	3	San Diego	0.881 ms
3	sdsc-gw.san-bb1.cerf.net	(192.12.207.9)	4	San Diego	1.944 ms
4	pos0-0-155M.san-bb6.cerf.net	(134.24.29.130)	4	San Diego	4.640 ms
5	atm6-0-1-622M.lax-bb4.cerf.net	(134.24.29.142)	4	Los Angeles	9.598 ms
6	pos6-0-622M.sfo-bb3.cerf.net	(134.24.29.233)	4	San Francisco	415.317 ms
7	pos10-0-0-155M.sfo-bb1.cerf.net	(134.24.32.86)	4	San Francisco	16.813 ms
8	192.205.31.29	(192.205.31.29)	6	New Jersey	16.917 ms
9	att-gw.sf.cw.net	(192.205.31.78)	4	San Francisco	81.281 ms
10	corerouter2.SanFrancisco.cw.net	(204.70.9.132)	4	San Francisco	81.254 ms
11	core1.Washington.cw.net	(204.70.4.129)	3	Washington	89.727 ms
12	mix1-fddi-0.Washington.cw.net	(204.70.2.14)	4	Washington	89.708 ms
13	**vsnlpoone.Washington.cw.net**	**(204.189.152.134)**	**4**	**Poone**	**706.301 ms**
14	202.54.6.17	(202.54.6.17)	6	Madras	697.946 ms
15	202.54.6.254	(202.54.6.254)	6	Madras	702.893 ms
16	giasmda.vsnl.net.in	(202.54.6.161)	4	Madras	704.856 ms

Figure 4: A sample traceroute output produced by the first phase of GTrace.

with the RTT values since they incorporate several components of delay. The RTT between two nodes has four components: the speed-of-light propagation delay, the amount of time it takes to transmit the unit of data, queuing delays inside the network and the processing time at the destination node to generate the ICMP time exceeded message. Traceroute typically sends 40-byte UDP datagrams, so it is safe to assume negligible transmit time. Ideally, for the verifier algorithm one would like the RTT to represent only the propagation delay, but this is not the case due to variable queuing and processing delays, hence it is not possible to set the upper bound on the RTT to a hop. Accordingly the verifier algorithm uses the minimum RTT returned by traceroute, as this would represent the best approximation of the propagation delay. Things are further complicated by the fact that the RTT delta between hops k and k+1 can be biased because the return path the ICMP packet takes from hop k can be totally different from the return path it takes from hop k+1. The Location Verifier tries to re-determine RTT values for hops it thinks are biased using ping.

By default, traceroute sends three datagrams each time it increments the TTL to search for the next hop. Changing the value of the q parameter in the GTrace configuration file will modify this behavior. The larger the value of q, the more accurate the estimate of the propagation delay, but large values of q also slow down GTrace as traceroute has to send q packets for each hop.

Knowing the geographical distance between two nodes, GTrace can calculate the time-of-flight RTT (the propagation delay at the speed-of-light in copper), compare it against traceroute's value and flag a problem if the RTT is smaller than physically possible. In such a case either the location of the source or of the destination or both is incorrect. The details of the verification algorithm are as shown in the next section.

Verifier Algorithm

1. Ideally, the RTT to hop k in a path should always be less than the RTT to hop k+1 or k+2, but this is not always true due to queuing delays, asymmetric paths and other delays. We allow a 1ms fudge factor to cover such discrepancies. Thus the RTTs between hops k and k+1 should be such that RTT(k) $\leq$ RTT(k+1) + 1 ms. If this condition does not hold true then the RTT to each of the out-of-order hops preceding hop k is estimated again with ping, i.e., till the first hop j preceding k such that RTT(j) $\leq$ RTT(k+1) + 1 ms. If the RTT estimates obtained using ping still do not satisfy the condition RTT(k) $\leq$ RTT(k+1) + 1ms, then hop k is not used in the later stages of the verifier algorithm.
2. Cluster the traceroute path into regions having similar RTT values. This is based on the assumption that nodes with similar RTTs will tend to be in the same geographic region.
3. For each region identified in the previous step, calculate the time-of-flight RTT for pairs of hops that have locations. If the RTT delta reported by traceroute for that pair of hops is smaller than the time-of-flight RTT, flag the pair of hops so that it is corrected in step 5.
4. Repeat step three for hops falling on the edges of adjacent regions.
5. Try to 'correct' unreasonable location values that were identified in steps three and four using the reliability of the search step that produced the location match. Adjacent nodes between regions are corrected first because they represent larger and probably more inaccurate locations. Correcting the nodes identified in step three follows this. By correct, we mean trying different alternatives for the incorrect location based on the cluster in which it falls, flagging it to the user and not plotting it in the display.

Example

Consider the trace shown Figure 4, where locations are expressed as city names for ease of illustration. The Search Step column indicates which step of the search algorithm produced the location for that hop. Step one of the verifier algorithm would mark hop 13 as unusable since its RTT is greater than its subsequent hops. In this case it is probably due to the return path from hop 13 being longer than that from hop 14. Next, step two of the algorithm would cluster the traceroute path into the following regions: 1-4, 5, 6-8, 9-10, 11-12 and 14-16. Step three would flag that there is a problem between hops 7 and 8 since it is not possible for a packet to travel from San Francisco to New Jersey in less than a millisecond. Likewise, step four would flag a problem between hops 10 and 11. Step five would first try to correct hops 10 and 11 since they fall in different regions. Seeing that the location for hop 11 was obtained through step three of the search algorithm and hop 10 was from a higher step, the Location Verifier would change hop 10's location to that of hop 11's, in this example to Washington and rerun the algorithm from step 3. This process is repeated until all locations from one hop to the next are physically realistic. In the end the Location Verifier would have indicated to the user that hop 8 is incorrect and is most probably located somewhere near San Francisco. Hops 9 and 10 are also incorrect and may be in Washington with their interfaces labeled San Francisco to identify the other end of that link.

Configuration Files

The configuration options in GTrace are quite flexible. How it functions and executes the search algorithm depends on the contents of two configuration files: GTrace.conf and GTraceMaps.conf.

GTrace.conf

GTrace.conf specifies the location of the commands GTrace uses and lists databases, text files, Lookup Servers if any, to use in the search algorithm. Figure 5 shows an example configuration file. This file is automatically generated by the configure scripts while installing GTrace.

Green	Both endpoints are authoritative locations.
Yellow	One endpoint is authoritative and the other is a guess whose location is not a country center, state center or obtained from a whois record.
Blue	Both endpoints are guesses and the locations of both the endpoints are not a country center, state center or obtained from a whois record.
Red	One endpoint is a location that is a country center, state center or obtained from a whois record.

Table 1: Reliability representation colors.

GTraceMaps.conf

The GTraceMaps.conf configuration file specifies attributes of the maps that GTrace uses in displays. Users can add their own maps as part of or independent from the existing world hierarchy. Independent maps allow users to describe their own intranet topology and then use GTrace as a graphical debugging tool within their network.

GTrace Features

Figure 6 shows an example of a trace that was executed from University of Colorado, Boulder to CAIDA in San Diego. On the display, the colors of the lines on the map indicate the reliability of the location obtained for the endpoints. The colors are decided based on the criteria in Table 1.

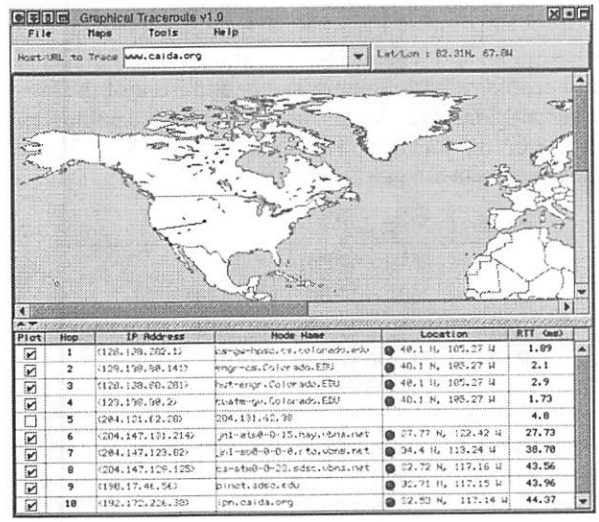


Figure 6: Example of a trace produced by GTrace.

The table in the lower section of the display consists of six columns. The first column provides the user with a checkbox that is enabled for each location plotted on the map. The user can disable a checkbox and the corresponding location will be skipped.

```
#GTrace configuration file

#Paths
TRACEROUTE=/usr/sbin/traceroute -q 3
WHOIS=/usr/bin/whois
PING= /usr/sbin/ping
NSLOOKUP=/usr/sbin/nslookup
DOMAINFILES=/home/ram/gtrace/data
DATABASES=/home/ram/gtrace/db

#Names of databases and text files to be used
#for location lookups. Order is important, list
#them in the order they should be searched.
CITIES=cities.db
AIRPORTS=airport.db

HOSTSLOC=Machine.db,hostnames/ipaddr;
         Hosts.db,ipaddr;
         Organization.db,hostnames/ipaddr;

TEXTFILES=England.txt,hostnames/ipaddr;

#Location of Lookup Servers if any
LOOKUPSRVS=
```

Figure 5: Sample GTrace.conf file.

Locations that are flagged as unreasonable by the Location Verifier are not plotted by default.

The second, third and fourth columns display the hop number, IP address and host name respectively. Clicking on columns three and four will bring up whois information for the node.

Column five provides the latitudes and longitudes obtained for each hop. Clicking on this column will provide an explanation of how the location was determined and whether the Location Verifier detected any problems. A small colored ball in front of the latitude and longitude value indicates which search step produced the location. The colors and the search step they represent are given below:

Green	Step 2 LOC record.
Yellow	Step 3 Complete match
Blue	Step 4 Domain parsing file
Cyan	Step 5 Hostname reduction match
Red	Step 6 whois record
Gray	Step 7 Country code

The last column shows the smallest of the round trip times returned by traceroute. The color of the value indicates how many packets timed out: black implies that no packets timed out, blue implies that one packet timed out, and a value in red indicates that two or more packets timed out.

Using GTrace in the Local Environment

System Administrators often use traceroute as a debugging tool to identify problems in their network. GTrace provides a visual representation that can facilitate understanding and debugging of their network. It can be used to discover routing loops as well as for deciding routes. For example in a large campus if a path from host A to host B (located in the same building) goes across campus and back, the routing could be fixed to avoid such inefficient paths. GTrace can also be useful from an end user perspective. Students can use the tool to work out the topology of their campus network.

Conclusion

GTrace is a handy tool for identifying network topology and routing problems as well as gaining more macroscopic insight into the Internet infrastructure. While GTrace uses several heuristics to determine locations and its approach does not guarantee accuracy, it is robust and extensible. New databases, new Lookup Servers and learned insights into ISP's naming conventions can easily be added to GTrace. We hope that users and system administrators will find GTrace useful and contribute their own domain parsing files, or even run their own Lookup Servers for community use.

The practical success of GTrace lies in the rules defined for the '.net' domains, since these comprise the majority of hops in many traceroutes. Looking up a '.net' name in the whois database is only useful for small localized ISPs. Relying on whois heuristics would result in backbone providers' '.net' nodes to all uselessly map to a single corporate headquarters for that provider.

The accuracy of this tool would be much improved if the Internet community maintained LOC records in the DNS. Unfortunately since LOC records are optional, non-trivial in effort to support and without any clear payoff to ISPs, pervasive use of them will probably never occur and geographic visualization of arbitrary Internet infrastructure will continue to require heuristics to determine physical location of nodes.

Acknowledgments

We would like to thank kc claffy at CAIDA for suggesting the idea to develop this tool. We would also like to mention a special word of thanks to the following people and institutions: VisualRoute for permission to use their maps and labels, Sleepycat Software for the BerkeleyDB Package, Jim Donohoe for developing NetGeo and to the entire research team at CAIDA who helped with many aspects during the development of GTrace.

Several students (Colorado: Robert Cooksey, Brent Halsey, Jamey Wood, Jeremy Bargen and UCSD: Jim Anderson) wrote graphical traceroute tools as class projects in Evi Nemeth's Network System's class. Many good ideas from these students' projects were incorporated into GTrace.

Availability and Support

GTrace-1.0 is the current release and it can be downloaded from the GTrace home page at http://www.caida.org/Tools/GTrace. The source code comes with the GTrace distribution. Further information on using the tool or how you can contribute domain parsing files can be found on the GTrace home page.

Author Information

Ram Periakaruppan is pursuing his Master's degree in Computer Science at the University of Colorado, Boulder. He can be reached at <ramanath@cs.colorado.edu>.

Evi Nemeth has been a computer science faculty member at the University of Colorado for years. Currently she is on leave doing the IEC (Internet Engineering Curriculum) project at CAIDA (Cooperative Association for Internet Data Analysis) on the UCSD campus and working furiously to make the publisher's deadline for the third edition of the UNIX System Administration Handbook. She can be reached at <evi@cs.colorado.edu>.

References

[AirportCodes] Listing of Airport Codes, http://www.mapping.com/airportcodes.html.

[BerkeleyDB] BerkeleyDB Package Distribution, http://www.sleepycat.com.

[DBCAIDA] Database files compiled by CAIDA, http://www.caida.org/NetGeo/NetGeo/.

[DBNDG] Database file compiled by NDG Software, http://www.dtek.chalmers.se/~d3augus t/xt/dl/.

[Halsey98] Brent Halsey, Visual Traceroute, Project Report submitted in Evi Nemeth's Networking class at University of Colorado, Boulder, 1998, http://www.caida.org/Tools/GTrace/paper/halsey.pdf.

[Jacobson88] Van Jacobson, Traceroute source code and documentation. Available from: ftp://ftp.ee.lbl.gov/traceroute.tar.Z.

[NetGeo] The Internet Geographic Database, http://www.caida.org/Tools/NetGeo.

[OROMatcher] OROMatcher – Regular Expression Package for Java, http://www.savarese.org.

[Peterson] Peterson, Larry L., & Davie, Bruce S., Computer Networks – A Systems Approach, Morgan Kaufmann, (1996).

[RFC1876] RFC 1876, Davis, C., Vixie, P., Goodwin, T., and Dickinson I., A means for Expressing Location Information in the Domain Name System, January (1996).

[RFC1918] RFC 1918, Rekhter, Y., Moskowitz, B., Karrenberg, D., Groot, G. J., Lear E., Address Allocation for Private Internets, February (1996).

[Swing] Java Foundation Classes – Swing http://java.sun.com/products/jfc/.

[VisualRoute] Maps from VisualRoute, http://www.visualroute.com.

[Wood98] Jamey Wood, *Graphroute*, Project Report submitted in Evi Nemeth's Networking class at University of Colorado, Boulder, 1998, http://www.caida.org/Tools/GTrace/paper/wood.pdf.

rat: A Secure Archiving Program With Fast Retrieval

Willem A. (Vlakkies) Schreüder – University of Colorado at Boulder
Maria Murillo – University of Colorado at Boulder

ABSTRACT

A new archive format called **rat** was developed. This format was designed to allow very fast retrieval of individual files. This is achieved using a table of contents to quickly find the file.

Each file in the archive is individually compressed with a compression method specific to the file. A user created configuration file is used to specify what type of compression to use on each file based on parameters such as the file extension and file size. Multiple sets of rules can be defined and activated from the command line to achieve different aims such as speed or size or to deal with different types of file sets. Parameters passed to the compression algorithms may also be specified.

The format also provides for signatures to be stored with the files. The program will generate and save the signature when the archive is created and verify the file when the archive is restored. Encryption is possible but not implemented.

The format is quite robust. If the archive is truncated or the table of contents is lost, the files in the portion that survived can still be recovered. Every file and table is preceded by a magic number so even recovery from bit rot may be possible.

The current implementation incorporates gzip (as zlib), bzip2, and LZO compression. Library versions of these compression algorithms are linked in for performance reasons. Only PGP signatures are currently implemented. Due to export restrictions on encryption software, a child process is spawned to execute a separate binary to do the signature creation and verification.

A library called **librat** implements all the functionality required to create the archive and restore files from the archive. Alternate user interfaces or embedded applications are therefore quite readily created. Three front ends to **librat** have been implemented. The first front end is a simple command line interface similar to **tar**. The second front end is character based interface that allows the user to browse the archive and selectively restore files similar to the **restore** program used with **dump**. The third front end is a GUI implemented using Qt.

Introduction

Archiving tools are widely used for many purposes. Of these **tar**, **zip**, and **dump/restore** are probably the most widely used. **dump/restore** is mostly for tape backups, although **restore** does provide a user interface to browse the archive and select which files to restore. **zip** is very popular on DOS based systems and is geared towards making archives on disk rather than to tape. **tar** is the most widely used archive program on UNIX systems. It is used for backing up to tape as well as making archives on disk.

As a class project for the System Administration class of Evi Nemeth at the University of Colorado at Boulder, a requirement for a public domain archive program that combine the best features of **tar**, **zip** and **dump/restore** was posed as a term project. The specific features that this program should have were defined as follows:

- The archive should use space as efficiently as possible.
- The archive should be geared towards fast retrieval of individual files.
- The format should be robust and allow access to files in case of errors such as a truncated archive.
- The format should support security.
- The format should be extensible.
- Special files such as hard links, soft links, files with holes and device files must be supported.
- The archive is intended to be written to a random access device such as a disk drive rather than a tape drive.

For the implementation, it was decided that the following features are important:

- The design should be modular to allow different front ends and embedded applications.
- Support for multiple, fixed-size volumes should be incorporated.
- Support for commonly used compression methods should be linked into the program for performance reasons.

- Due to export restrictions, any encryption software should be invoked as an independent binary.
- The implementation should support remote backups.
- A front end with a **restore**-like browsing facility should be provided.

Fulfilling the above requirements was determined to be more important than being compatible with existing software like **tar**.

Much consideration was given to adapting existing software like **tar** to the task. **tar** is a de facto industry standard, but has been superseded by **pax** in the POSIX.2 specification. The new, extended file format for tar is specified in POSIX.1. **tar** is freely available, and the Gnu tar-1.12 implementation already supports device files, files with holes, remote backups and such desirable features.

One approach would be to implement per-file compression in tar, and then store the table of contents as just another file. When restoring the archive with an older version of tar, the files would be restored as the compressed version of the file which can then be manually uncompressed. The table of contents would be restored as an extra file. While this is a valid alternative, this approach was abandoned because the tar format does not use space efficiently. **tar** archives are written in 512 byte blocks. The header for each file uses 512 bytes, and the data are stored in 512 byte blocks. A one byte file would therefore consume 1024 bytes. The blocks are padded with null bytes. When the entire archive is compressed using gzip, these inefficiencies are less important because long runs of null bytes compress well. However, the drawback of this approach is that the entire archive must be filtered through the decompression algorithm to retrieve any individual file.

Header
File*0*
...
File*n*
Table of Contents
UID Table
GID Table

Figure 1: Rat archive format.

In order to fulfill all the requirements set out above, a completely new archive format named **rat** was designed and implemented. The name **rat** was chosen to be **tar** backwards, and to fit in with the animal theme of some of the names in the GNU project and book covers from O'Reilly and Associates.

rat Archive Format

Figure 1 shows the layout of the **rat** archive. All strings in the archive are saved as variable length sequence of bytes terminated by a null. All integers are saved as unsigned binary integers with the most significant byte first.

The header section contains the archive label and similar global information, as well as 64 bit pointers to the start of the files section, table of contents (TOC) and UID and GID tables. The order in which the remaining sections of the archive appears is determined by these pointers, but this is usually in the order shown in Figure 1.

Magic	(uint32)
Header Size	(uint16)
Signature Type	(uint8)
Compression Type	(uint8)
Mode	(uint16)
UID	(uint32)
GID	(uint32)
Mtime	(uint64)
Ctime	(uint64)
Uncompressed Size	(uint64)
Flags	(uint16)
Flag Dependent Data (ACLs,...)	
Data Size	(uint64)
Data	
Signature Size	(uint16)
Signature	

Figure 2: File Layout.

Files

The files section is the bulk of the archive. It consists of files and file pointers saved contiguously. A file pointer contains the file name, file type and a pointer to the file. A file contains the data for the file as shown in Figure 2. Each file contains all the relevant information from the inode of the file, the contents of the file (data blocks) and possibly a signature.

This layout closely mirrors the layout in the disk file system. The file pointers correspond to directory entries. Hard links are supported by having multiple file pointers point to the same file. The file pointer names the file, but does not contain any data other than the file type and a pointer to the file. The special value zero in the pointer field is used to indicate that the file immediately follows the file pointer.

The file entry contains the file data but not the name of the file. The file data consists of three sections, a header section, data section and signature section. Each section is preceded by a size entry, indicating the size of the rest of the section. This makes it easy to process the contents of the file by reading only the size entries and treating the remained of the section as a block of data. The data size is stored as a 64 bit integer so extremely large files are supported.

The header section contains file attribute information such as the owner, permissions, size and times,

but not the name of the file. Times are stored as 64 bit integers in the UNIX style. Special information such as the device major and minor numbers and soft link text are stored in the data section. The data size will always reflect the true number of bytes used by the data section.

Only the UID and GID are stored for each file. The corresponding user and group names can be obtained from the UID and GID tables stored in the archive. The Flags field is used to signal various special cases. Bit 0 in the Flags field is used to indicate that the file contains holes. Such files are stored like any other file since the compression algorithms compresses long sequences of zeroes quite efficiently. When this flag is set, the restore function finds long sequences of zeroes and recreates the holes. Bit 1 in the Flags field is used to indicate that the file has an associated Access Control List (ACL) which will appear at the end of the header. The remaining bits are reserved for future expansion.

Code	Algorithm
0	None
1	gzip (zlib)
2	bzip2
3	LZO

Table 1: Compression Codes.

The Compression Type field defines the compression method used in the data section. Table 1 shows the codes used for the different methods currently implemented. The Signature Type field defines the type of signature used. Currently only values of 0 (CRC-32), 1 (MD5) and 2 (MD5+PGP) are implemented. This field can also specify an encryption method.

The signature section contains the checksum of the header and data sections. The checksum always appears first in the signature section. When files are signed using PGP, the PGP signature immediately follows the checksum.

It should be noted that the file pointers in the file section replicates the information already contained in the table of contents (TOC) described below. The purpose of the file pointers is to allow recovery of the information in the archive if the TOC is missing, such as when the archive is accidentally truncated.

Table of Contents

The Table of Contents (TOC) typically follows the files section. The layout of the TOC is shown in Figure 3. For each file, the TOC contains only the file name, file type and two 64 bit pointers, one to the file pointer and one to the beginning of the actual file, both of which are in the files section. When the item in the TOC is a directory, the full path to the file is saved as the name. For all other items such as regular files, links and device files only the name of the file is stored. The full path is implied by the preceding directory in the TOC. The local directory is implied at the beginning of the TOC. In order for this to uniquely name each file, subdirectories must always appear after non-directory items. Reconstructing the full path name is trivial if the TOC is read sequentially. When the paths are long, this convention reduces the size of the TOC considerably.

Magic			(uint32)
Signature Type			(uint8)
Number of Entries			(uint64)
Name*0* (...0)	Type*0* (uint8)	Link*0* (uint64)	File*0* (uint64)
	...		
Name*n*	Type*n*	Link*n*	File*n*
Signature Size			(uint16)
Signature			

Figure 3: Table of contents.

The purpose for saving both the offset of the file pointer and the file itself in the TOC is to expedite updating of the archive. The pointers are used to track which file pointers and files are still referenced. Unreferenced items are removed when the archive is updated.

A checksum of the TOC is computed and stored in a signature section identical to that for a file in the files section. If so indicated by the signature type code, the checksum will also be signed.

The layout of the UID and GID tables are similar to the TOC. It lists the UID and user name or GID and group name in pairs, ordered by UID or GID. These tables are typically small, but can save a large amount of space since the user and group names does not have to be saved for each file. Both the UID and GID tables have corresponding signature sections.

Security

The **rat** archive format attempts to make the archive secure by securing each individual file, the TOC, UID and GID tables. A checksum is computed for every such entity. Currently CRC-32 (32 bit) and MD5 (128 bit) checksums are implemented, but more secure checksums can be added as they become available. The type of checksum can be set in the **.ratrc** configuration file, the default being MD5.

Of course, checksums only protect the archive from accidental corruption, not malicious alteration. To secure the archive, the checksum can be signed. In the current implementation, only the use of PGP to sign the archive is supported, and implies that an MD5 checksum will be used.

When a file or the TOC, UID or GID table is read from the archive, the checksum is recomputed and compared against the stored checksum. If signed, PGP is used to verify the checksum. If either the

checksum does not match the stored checksum, or the signature does not verify the checksum, the read will return the result ErrCheck.

librat Implementation

The **rat** archive was implemented as a library named **librat**. The **librat** interface is quite compact. The calls to **librat** to open and close the archive are:

```
rat_t rat_init(char* device,
      int verbose,
      char* label,int level,
      char* user,char* pass);
int   rat_open_write(rat_t rat);
int   rat_open_read(rat_t rat);
int   rat_open_update(rat_t rat);
int   rat_close(rat_t rat);
```

A call to rat_init is used to initialize the library and initialize parameters such as the device to be used for the archive, verboseness level, set of rules (level) and user name and pass phrase for the signature.

rat_init reads the **.ratrc** configuration file and returns a private structure of type rat_t that is used by all other functions.

When creating an archive, rat_open_write is used to open the archive, while rat_open_read is used to open the archive for reading. To modify an existing archive, rat_open_update is used to open the archive. When done, rat_close is used to close the archive.

Reading an archive is achieved by calling the functions

```
int   rat_open_toc(rat_t rat);
int   rat_next_toc(rat_t rat,
        char path[],
        char* type,
        u_int64_t* offset);
int   rat_inq_file(rat_t rat,
        const u_int64_t offset,
        info_t* info);
int   rat_read_file(rat_t rat,
        u_int64_t offset);
int   rat_check_toc(rat_t rat);
```

Each call to rat_open_toc starts reading the TOC from the beginning. Each call to rat_next_toc returns the full path name of the next entry in the TOC as well as the type and offset. The returned type can be tested using an set of enumerated constants defined in the header file. The offset returned by rat_next_toc can be passed to rat_inq_file or rat_read_file. A call to rat_inq_file returns all the fields stored in the file header in an structure of type info_t. A call to rat_read_file restores the file, including any necessary directories. If the file has a signature, rat_read_file also checks the signature. Note that rat_read_file will create missing intervening directories, but will not recursively restore files and subdirectories.

The integrity of the archive can be checked by calling rat_check_toc. Not only does this call check that the TOC matches file pointer entries in the files section, but it also checks the checksums and signatures of the files.

After calling rat_open_write or rat_open_update, a file can be added to the archive by calling

```
int rat_add_toc(rat_t rat,
                const char* path);
```

A call to rat_add_toc simply adds the list of files specified to the TOC. Only when rat_close is called are the files in the TOC written to the archive. rat_add_toc recursively adds all files and subdirectories when the path is a directory. rat_add_toc returns -1 if the addition failed, typically because the file does not exist or is not readable. If the addition succeeded, rat_add_toc returns the number of entries in the TOC that was replaced. Adding all new files will result in a 0 being returned. The existing entries in the TOC are searched for every file being added to ensure that each file is unique. Adding the same file twice causes the previous occurrence to be replaced.

After calling rat_open_write or rat_open_update, a file can be removed from the archive by calling

```
int64_t rat_find_toc(rat_t rat,
                     const char* path);
int rat_del_toc(rat_t rat,
                u_int64_t index);
```

A call to rat_find_toc is used to get the index of the named file in the TOC. If the file is not in the TOC, -1 is returned. Both files and directories are found by rat_find_toc. A call to rat_del_toc with the index returned by rat_find_toc deletes that file from the TOC. If the index points to a directory, all files and subdirectories of that directory are also deleted from the TOC. As with rat_add_toc, removing the files from the archive is delayed until rat_close is called.

A call to rat_close closes the archive. If the archive was opened with rat_open_read, the archive is simply closed. If the archive was opened with rat_open_write, all the items in the TOC are written to the archive before it is closed. If the archive was opened with rat_open_update, a call to rat_close will cause the archive to be updated to reflect the current contents of the TOC. When rat_open_update is first called, the TOC, UID and GID tables are read into memory. Subsequently, when rat_close is called, the first operation is to remove all items in the archive that were removed by calls to rat_del_toc or replaced by calls to rat_add_toc. This is done in place, without copying the archive, by moving the remaining entries in the archive closer to the head of the archive so that the parts of the archive that did not change are contiguously packed at the head of the archive. New files are then appended to the truncated archive. Finally, the TOC, UID and GID files are appended from memory. This procedure makes adding only new files efficient since it requires only the TOC, UID and GID tables to be read from the archive, the files added, and then the

TOC, UID and GID tables rewritten to the archive.

Calls to rat_open and rat_close can be freely intermixed. The library will flush changes to the archive as necessary. For example, after creating a file with calls to rat_open_write and rat_add_toc, a call to rat_open_read will write all files to the archive and reopen the archive for reading. Extra calls to rat_close will return safely.

Finally, rat_free should be used to free the rat structure allocated by rat_init.

The .ratrc Configuration File

When rat_init is called, **librat** attempts to read a file called **.ratrc**. It first tries the current directory for a **.ratrc** file, then the user's home directory for a **.ratrc** file, and finally /etc/ratrc. The first file found is used.

The purpose of the **.ratrc** file is to allow the user to configure **librat**. In particular, **.ratrc** is intended to control the compression algorithms to be used on particular files and how to interface with the security programs. An example **.ratrc** file is shown in Figure 4.

```
#  How to run pgp
pgps  pgps -btu%s %s -o-
pgpv  pgpv +batchmode=1 %s.sig
#  Global parameters for BZIP2
BZ2blockSize 9
BZ2workFact  0
#  Set of Rules 1
level 1
any CompNone
#  Set of Rules 2
level 2
any CompGZ
#  Set of Rules 3
level 3
any CompBZ2
#  Set of Rules 4
level 4
any CompLZO
#  Set of Rules 9 (default)
level 9
ext      .rat    CompNone
ext      .gz     CompNone
ext      .bz2    CompNone
ext      .gif    CompNone
ext      .jpg    CompNone
hole             CompBZ2
smaller 8192     CompGZ
any              CompBZ2
```

Figure 4: Example **.ratrc** file.

When calling rat_init, the level parameter is interpreted as the compression level. In the **.ratrc** file, this is manifested as up to nine sets of rules named 1 through 9, each corresponding to a compression level. Only the rules outside any set of rules and the selected set of rules (level) are evaluated. Rules are evaluated in the order they appear in the **.ratrc** file. The first rule to match is used to select the algorithm to use for each file.

The user can configure these sets of rules to reflect local experience with the types of files being archived. Different sets of rules can be tuned for speed, archive size, or any such desired metric. The **.ratrc** file shown in Figure 4 defines five such sets of rules that were used to generate the results shown in Table 3 below. Set 1 causes all files to be stored without compression. Sets 2, 3 and 4 causes all files to be compressed using zlib (as in gzip), bzip2, and LZO, respectively.

Set 9, which is the default rule set, uses a non-trivial set of rules. Files for which the file name ends in .rat, .gz, .bz2, .gif and .jpg are assumed to be already compressed, so attempting to re-compress them is futile. Therefore these files are stored in the archive without further compression. The bzip2 algorithm in general does better with big files than the gzip algorithm, while the opposite is true for small files. Therefore Set 9 uses the gzip algorithm to compress files less than 8192 bytes in size, while the bzip2 algorithm is used for all other files. The bzip2 algorithm is also used for files with holes.

Some compression algorithms are themselves configurable. For example, Figure 4 shows the use of the BZ2blockSize and BZ2workFact parameters which are used to control the bzip2 algorithm. Since these parameters are specified outside any set of rules, they apply to all sets. However, different values could be specified inside the rule set which would then take precedence. Using such configurable parameters allows the user to tune the algorithms to best suit the local conditions.

Finally, the example in Figure 4 also controls how to invoke PGP on the local system. The particular version shown is for PGP 5.0i. The pgps line controls how the signature is generated, while the pgpv line controls how the signature is verified. The parameters filled in on the pgps line are the user name and the file name, while only the file name is filled in on the pgpv line.

Example Implementations

Three example implementations were written. The first implementation is a simple command line style interface similar to tar. This implementation required 244 lines of code (including comments) of which only about 50 lines actively manipulate the archive.

A more elaborate implementation that allows the user to browse the archive was also written. In this implementation familiar commands such as ls, pwd and cd are used to browse and traverse the archive. The header command is used to list all the information

about a file. Commands such add and delete are used to add or remove files to or from a list of files to restore. The command lsmark is used to show the list of files marked for extraction, while the extract command is then used to restore all files marked for extraction to disk.

Finally an implementation with a graphical user interface (GUI) was written using the Qt library. This implementation shows the files in the archive as a tree structure with boxes next to each file name. Boxes can be checked to specify that files are to be extracted. Clicking on the box next to a directory marks or unmarks all files in the directory and subdirectories. This implementation required only 390 lines of C++ code. Figure 5 shows a screen shot of this implementation.

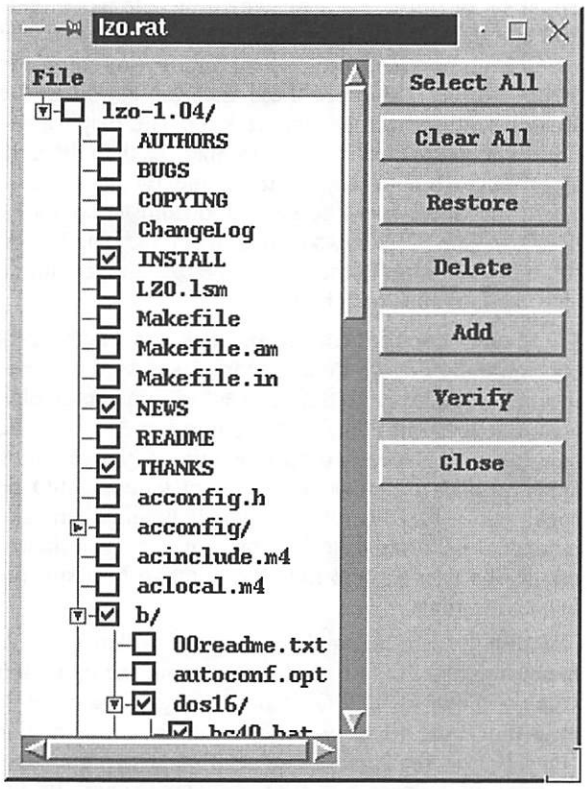


Figure 5: Qt rat.

All the implementations use calls to rat_open_toc and rat_next_toc to build a tree structure of the TOC. For detailed file listings calls to rat_inq_file are used. To translate the UID and GID to user and group names, calls are made to

```
const char* rat_inq_uid(rat_t rat,
                        u_int32_t uid)
const char* rat_inq_gid(rat_t rat,
                        u_int32_t gid)
```

Once the user has selected a list of the files to extract, repeated calls are made to rat_read_file to restore the files.

Results

The **rat** implementation was tested on a mixed data set. Statistics for the data are shown in Table 2. The total size of all the files is 236.5 MB. Note that the standard deviation is about ten times the average file size, and the largest file is 36 MB. The files are a mix of C source code files, object files, ELF binaries and postscript files. Tables 3a and 3b compares the results obtained with **rat**, **tar** and **zip**.

With no compression, the **rat** archive is only 0.3 MB larger than the total file size for 3186 files, while the **tar** archive is 2.2MB larger than the total file size. Creating the archive and restoring the entire archive takes a bit longer with rat, but **rat** retrieves a single file **much** faster than **tar**. It is clear that the **rat** file store and retrieve functions need to be optimized.

When using gzip, bzip2 or LZO to compress the archive the **rat** and **tar** results are very similar as far as the archive size, creation time and full restoration time are concerned. Support for gzip is incorporated into tar. bzip2 compression is achieved by piping the output through the bzip2 utility, while decompression is achieved by processing the output from the bzcat utility. LZO compression and decompression is similarly achieved by piping through the lzop utility. When using gzip, bzip2 or LZO to compress the entire **tar** archive, the compression is theoretically more efficient than if each file is compressed individually as in **rat**. This is borne out by Table 3, but the increase in size is only a few percentage points. In all cases, however, almost instantaneous access to an individual file is achieved with **rat** while **tar** can take several minutes.

The **zip** utility uses an algorithm very similar to gzip, but also does per file compression. Table 3 shows that the resulting file size is slightly larger than the **rat** file, but better performance than either **tar** or **rat** when creating the archive. When restoring a single file, the performance of **rat** and **zip** are similar, but **zip** is slightly faster than **rat** when restoring all the files.

Note that the mixed set of rules that uses gzip on files less than 8192 bytes and bzip2 on larger files yields a slight improvement in archive size. Defining optimal sets of rules for file size remains a task for the future.

Finally, the data was archived and signed using a 1024 bit PGP key. This adds 66 bytes per file so the resulting archive is only slightly larger than the archive without signatures. However, creating and checking the signatures takes a considerable amount of time. This extra time is attributable to both the actual work of creating or checking the signature as well as spawning a new process to run PGP for every file.

Adding a single new file to the archive with **rat** takes less than a second and does not involve copying the archive. Adding a single new file to a **zip** or **tar** archive takes quite a bit longer, probably because of

the time it takes to copy the archive. A file cannot be added to a compressed tar archive. The whole archive must first be uncompressed, the file added, and then the whole archive recompressed.

Deleting a single file from the archive with **rat** or **zip** takes approximately the same time and is quite fast. **tar** does not support deleting a file a file from the archive.

Conclusions

The three sample implementations showed that **librat** is easy to use and provides sufficient functionality to construct a sophisticated user interface. The sets of rules in the **.ratrc** file proved to be very useful during the performance tests.

Number of Files	3186
Total Size	236.5 MB
Average Size	76.0 kB
Std.Dev. Size	749.6 kB
Maximum Size	36.1 MB

Table 2: Test Data.

The results shown in Table 3 and other tests performed with this implementation proved that archives could be produced that are comparable in size to those with **tar** or **tar** and **gzip**, **bzip2** or LZO for about the same amount of effort on the archive creation time, but with dramatically improved extraction speed for individual files. In terms of performance, rat compares well to **zip**.

The ability to sign and verify files proved to be very useful and adds little to the file size but is rather slow. The major advantage of signatures over encryption proved to be that usable files can be extracted even when signature verification software is not available.

Future Work

The **.ratrc** file allows very flexible specification of the compression method to be used on each file. A good heuristic to select a near optimal set of rules for general use remains to determined. It is likely that the type of rules may have to be expanded with such a heuristic. In particular, the rules should be expanded to check the file's magic as done by the file utility.

A major drawback of externally executing the signature creation and verification program is the poor performance. If the signature code (as opposed to the encryption code) could be build directly into **librat**, this should significantly alleviate the performance problems.

Acknowledgements

The authors wish to thank the instructor and teaching assistants for the System Administration course at the University of Colorado at Boulder during the Fall of 1999, Evi Nemeth, Vega Paithankar, Josh Prismon and Ali Rayl for their support, and Rob Braun for suggesting and guiding the project.

Availability

The initial distribution version of **rat** is rat-0.1. It consists of **librat**, the simple command line front end and the Qt front end. It is available for download from http://www.cs.colorado.edu/~vlakkies/.

Program	Compression	Size (MB)	Build Time (sec)	Extract One File (sec)	Extract All Files (sec)
tar	none	238.7	39	23	65
rat	none	236.8	74	<1	80
tar	gzip	77.7	262	27	62
rat	gzip	78.7	294	<1	72
zip	deflate	78.8	222	<1	67
tar	bzip2	67.3	1011	186	206
rat	bzip2	72.3	1090	<1	210
tar	LZO	104.3	61	16	75
rat	LZO	105.5	89	<1	69
rat	mixed	72.0	1064	<1	209
rat	gzip+PGP	78.7	10861	1	1689

Table 3a: **rat**, **tar** and **zip** performance.

Program	Add (sec)	Delete (sec)
tar	38	N/A
zip	24	15
rat	<1	7

Table 3b: **rat**, **tar** and **zip** performance.

Author Information

Vlakkies Schreüder is a consulting engineer, software developer and system administrator with Principia Mathematica, and a full time student. He holds a Ph.D in Computational Fluid Dynamics from the University of Stellenbosch and is currently working on a second Ph.D in Parallel Systems at the University of Colorado at Boulder. Contact him at <vlakkies@colorado.edu>.

Maria Murillo holds an MS in Geophysics from the University of Tulsa. She is currently a full time student pursuing a Ph.D in Computer Science at the University of Colorado at Boulder. Contact her at <murillo@colorado.edu>.

References

J. Seward, bzip2, a block-sorting file compressor, http://www.bzip2.org.

J. Gailly and M. Adler, GNU Zip, http://www.gzip.org/.

M. F. X. J., Oberhumer, LZO – a real-time data compression library, http://wildsau.idv.uni-linz.ac.at/mfx/lzo.html.

P. Zimmerman, Pretty Good Privacy, http://www.pgpi.com/.

AT&T, tape file archiver, http://ftp.digital.com/pub/GNU/tar/.

M. Adler, R. B. Wales, J. Gailly, G. Roelofs, O. van der Linden and K. U. Rommel, Info-ZIP, http://www.cdrom.com/pub/infozip/.

J. Gailly and M. Adler, zlib, a general purpose data compression library, http://www.cdrom.com/pub/infozip/zlib/.

Cro-Magnon: A Patch Hunter-Gatherer

Jeremy Bargen – University of Colorado at Boulder and Raytheon Systems Company
Seth Taplin – University of Colorado at Boulder and CiTR, Inc.

ABSTRACT

On a relatively large and heterogeneous network, there may be several operating systems and dozens of major applications in general use. Locating and maintaining patches for these systems can take up a significant portion of a system administrator's time. In addition, groups of machines must all be kept at consistent patch levels, and the exact patch level may vary depending on the group. Security patches are especially problematic because they appear at irregular intervals, and the administrator generally wants to find and install them as soon as possible after they become available.

This paper describes Cro-Magnon, a system for automating the process of patch downloading and application. Cro-Magnon can be configured with a list of patch sites and will mirror those sites, downloading new patches as they are detected and notifying the administrator of the downloads. Cro-Magnon can verify patch authenticity and can maintain patch data for multiple machine groups and architectures, all with different administrators.

The Cro-Magnon architecture is intended to be as flexible as possible. It allows for multiple download methods such as FTP and HTTP and multiple authentication schemes like MD5 and PGP. Although it currently deals primarily with patch downloading and notification, it is intended to be extended to allow automated patch application and maintenance.

Introduction

System administrators must often maintain large networks of heterogeneous systems. In order to keep the network as secure as possible, it is important that the system administrator be aware of the latest versions of all security patches as soon as they become available. In addition, administrators must often keep operating systems as well as applications up-to-date with the latest bug fixes. In some cases (especially for software development environments), the administrator must even maintain all machines in one logical group at one consistent patch level, while machines in another group are maintained at a different level.

For a large network, the amount of work involved in patch management can quickly become overwhelming. System administrators may not have time to locate and download security patches until days or weeks after a security advisory is first issued. This can cause periods of dangerous insecurity for the network. In addition, when the network is composed of multiple types of operating systems, each with several patch sites, it becomes easy to miss some sites altogether, so that some fixes are never installed at all.

Cro-Magnon is a system that automates the process of patch gathering in two ways: first, it will search a specified list of Internet sites for patches and automatically download, verify, and notify the administrator about the patches. This enables the administrator to find out about new security and bugfix updates promptly and reduces the chance that any one site will be overlooked. Second, Cro-Magnon is ultimately intended to simplify the process of applying patches and distributing them to select groups of machines. This paper describes a full (although simple) implementation of the first goal.

Functional Overview

Cro-Magnon has two distinct modes of operation. The primary mode is as a daemon, for periodic background checking and notification of new patch status. Cro-Magnon can also run in an interactive mode, giving a "users-eye" view into the current state of the configured systems and applications and their available patches. Downloaded patches are stored in a hierarchical layout in the filesystem, separated by system architecture and applications.

In daemon mode, Cro-Magnon detaches from the shell and runs in the background, periodically waking up and checking to make sure that its patch database is up to date with the latest patches available for its configured systems and applications. If a new patch is found, it is automatically downloaded into the filesystem, optionally verified via MD5 or PGP signature file (as defined by the configuration) and the system administrator is notified (typically by email) of the patch's arrival. This mode allows a system administrator to start Cro-Magnon from a startup script and let Cro-Magnon automate the generally routine job of checking vendors and distribution sites for the latest patches.

Interactively, Cro-Magnon provides a simple console-based user interface so that a system administrator can examine the current state of the configured systems. The user interface provides the rudimentary

ability to view the known systems, machine groups, and applications that are currently monitored, and the various patch levels that Cro-Magnon has detected for them.

System Architecture

The Cro-Magnon system is composed of a core "Engine" module, written in Perl, surrounded by various plug-in Perl modules (Figure 1). Most of the functionality of the system is supplied by the plug-ins, including patch downloading, patch verification, administrator notification, and the user interface. The goal of this design is to allow customized behaviors for any or all of the basic functions of the system, simply by writing modules that follow a documented API. The only functions in the Engine are those which wire the plug-ins together using the configuration file and the main event loop for the daemon mode.

The configuration file divides the network up into "Systems," which are logically distinct hardware and/or software architectures; "Applications," each of which may represent a single application, an application suite, or the operating system itself; "Sites," which are locations such as FTP and Web sites from which patches may be downloaded; and "Groups," which associate a list of machine host names with a list of Applications, each of which may have a version number associated with it, so that different versions can be tracked for different groups or systems.

Each Application is associated in the config file with a number of Sites, each of which is associated with both a download module that tells it how to retrieve patches from that Site and a verification module that tells it how to verify the patches once they have been downloaded. Examples of possible download module functionality include anonymous FTP, FTP using a certain "registered user" account, HTTP, or even copying files from another directory on the same system. The most general approach is to retrieve every file in a directory, however, modules can be specialized to download specific files or filetypes as each situation requires. Examples of verification modules are MD5 or PGP verification, or "Simple" (no verification).

All patches are stored in a hierarchically structured section of the filesystem. The patch root directory is defined in the configuration file; one subdirectory exists under this root for each defined System. In a System's directory, each Application has a subdirectory. This subdirectory in turn contains one directory for each Site containing Application patches. This allows patches to be stored indefinitely for each remote site with no danger of overlap.

When running in daemon mode, Cro-Magnon will run in the background and will periodically wake up and download updates from the Sites it knows about. The system may be run as a daemon, in which case it will wake up periodically after an interval specified on the command line; or it may be told to run only once via a command-line flag. In the latter case, it is assumed that a tool like cron will be used to run Cro-Magnon periodically. This approach gives the end user maximum flexibility as to how the system is run. The download strategy is defined by the individual download module but should generally follow a "FTP mirror" strategy: a file is downloaded if it is new or if it has changed since the previous download of that file. If any files are downloaded from a specific Site, they will be verified using the Site's verification module. A download report listing the files downloaded, the Site they were downloaded from, and their new location in the patch database will then be generated and sent to the Group administrators interested in those files.

In interactive mode, Cro-Magnon functions as a patch viewer. The user can specify the exact System,

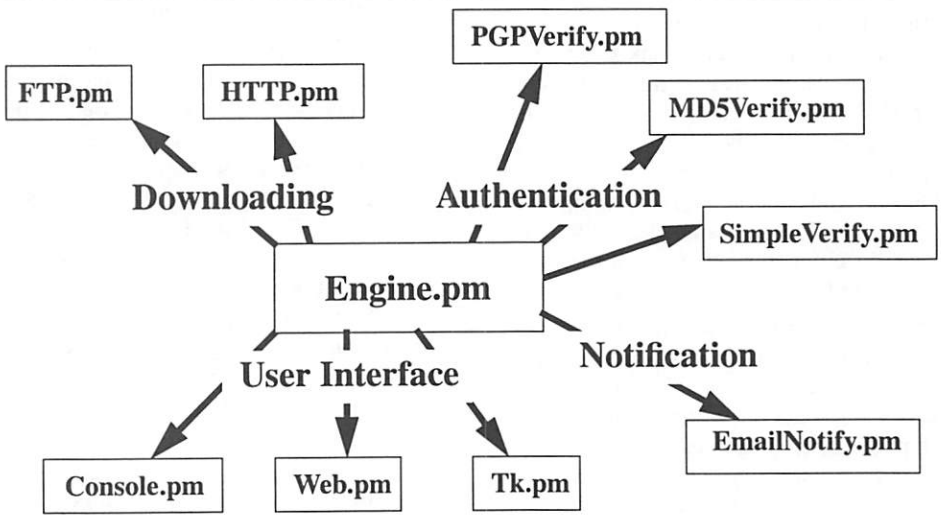


Figure 1: The Engine and sample plug-ins

Group, and Application of interest, and may then see all the downloaded patches for the application and Cro-Magnon's best guess as to which of them are installed. This 'best guess' is dependent on the ability of the software to determine the current version of a given application that has been installed, directly through Cro-Magnon, or possibly from additional information provided by the operating system for "manual" updates. Since there is little consistency in version identification across applications and operating systems, a version module must be used to return the current version of an application. For example, the Linux kernel version can usually be found using 'uname -rv' while perl and gcc both support a '-v' flag. In addition, interactive mode serves as a framework for many of the desired future extensions to the system (see "Future Enhancements").

Software Architecture

For any system administrator who has had to deal with a heterogeneous network, a primary concern for a "universal" tool is portability. To maximize the portability and extensibility of the Cro-Magnon system, it has been designed in the Perl language using extensible modules that provide a skeletal API framework. This framework, coupled with a flexible configuration file format, allows implementation modules to be specialized for unique environments.

The config file is built up from hierarchical keywords which define various records that describe the user's system, groups and applications. At the highest level, the "System <system name>"/"EndSystem" keywords delimit a system description, and "DefaultSystem <system name>", "DefaultGroup <group name>" and "DefaultApplication <application name>" keywords specify which system, group, and application (if any) will be selected as the defaults for interactive behavior.

An Application record is delineated by the "Application <application name>"/"EndApplication", and specifies a number of sites to check for

```
System   TestSys
   Application   TestApp1
      Site TestSite1
        Location   ftp://ftp.h2net.net/pub/nvb/test
        Location   ftp://ftp.somewhere.com/pub/mirrors/h2net/nvb/test
        ToDownload FTP.pm
        ToVerify SimpleVerify.pm
      EndSite

      Site TestSite2
        Location   ftp://ftp.freebsd.org/pub/FreeBSD/CERT/patches/SA-97:06
        ToDownload FTP.pm
        ToVerify SimpleVerify.pm
      EndSite
   EndApplication

   Application   TestApp2
      Site TestSite3
        Location   ftp://metalab.unc.edu/pub/gnu/grep
        ToDownload FTP.pm
        ToVerify SimpleVerify.pm
      EndSite
   EndApplication

   Group TestGroup1
      Machines
         saclass saclass-5
      EndMachines
      CurrentVersions
           TestApp1 Version-v saclass
           TestApp2 Version-RCS saclass
      EndVersions
      Administrator jbargen@acm.org
      Administrator taplin@cs.colorado.edu
      ToNotify EmailNotify.pm
   EndGroup
EndSystem

DefaultSystem TestSys
DefaultGroup TestApp1
DefaultApplication TestGroup1
```

Figure 2: Sample Config File

patches. Each site record is offset by a pair of "Site <site name>"/"EndSite" tags, and is composed of one or more locations designated by a "Location <URL>" entry (multiple locations designate multiple mirror sites), the module to be applied for the download mechanism "ToDownload <module name>", and the module to be applied for verification of the trustworthiness of a patch (for example MD5 checksum verification) denoted by the keyword pattern "ToVerify <module name>".

A Group record is bounded by a pair of "Group <group name>"/"EndGroup" keywords, and contains a machine list between a set of "Machines"/"EndMachines" tags, one or more "Administrator <email address>" keyword pairs, and a list of applications and versions that are currently installed. This application list is formatted between "CurrentVersions"/"EndVersions" tags, with an application name followed by the name of (and arguments to) a module which will return the current version. The version information is intended to be used to determine if a newly-found patch is of interest to the application, narrowing the selection criteria for notifying the appropriate administrators. The type of notification is specified by a ToNotify keyword.

The Engine can run autonomously as a daemon, checking sites for new patches, or interacting with a specified Display component, for example, a Console display. Display interaction is done through a callback mechanism, to allow for the development of graphical interfaces in the future. This opens up the possibility of specialized Displays that interact with the Engine through a Perl/Tk GUI, or even a Java GUI, which could be embedded in a web page for remote administration over a secure network connection.

Major Software Components

Each of the components of the CroMagnon system is composed of one or more Perl modules. All of the modules except for the Engine may be overridden by derived modules.

Engine

The Engine module is a stand-alone module providing the core functionality of the system. It is the only module that may not be overridden by a derived class. It is the glue that binds the rest of the modules together via the config file. The Engine's primary functions are parsing the config file to understand the Systems, Groups, and Applications that are specified, knowing (from the configuration) which modules to delegate to when dealing with each of these specifications, and whether to run in daemon mode to find and gather patches, or to interact with a user.

The initialization and reading of the config file is common to both operational modes, but the behavior of the Engine changes dramatically depending on the mode in which it is run. Running in daemon mode causes the Engine to detach from the console and periodically wake up and check for new patches. It processes each System in turn, finding and downloading new patches. The Engine then examines each Application's new patches and determines if there is a Group that is interested in that patch. If so, the Engine notifies that Group's administrators that the patch has been downloaded, verified, and is available for installation from a given directory. Sending a SIGHUP to the daemon causes it to re-read and re-parse the configuration file to pick up any recent changes, and then continue on its merry way.

In interactive mode, the Engine interacts with a Display component, providing configuration and patch information as described in the following section.

Display

The display component provides a user interface to the Cro-Magnon system. Although there is not much a user can currently do interactively, this serves as a placeholder for the functionality described later under "Future Work." The only user interface that is currently provided is a simple command-line display, but the architecture will support any interface that can be set up to use an "event loop" style of operation, including a conventional GUI using a package such as Perl/Tk, or even a separate process communicating with the Engine via a socket stream.

The display component interacts with the Engine by use of a dual-callback mechanism. The display runs an event loop which waits for user input. The user enters a request, which causes the display component to call a function in the Engine. The Engine does whatever processing is required, and invokes a callback function in the display that does whatever is required to inform the user of the Engine's actions. When the callback function returns, the display component goes back into its event loop.

In order to create a new display module, an implementor must override the Display abstract base class. There are a number of functions which must be overridden in this class, most of which are callbacks from the Engine. The display architecture does not specify anything about its interaction with the user; this is left entirely up to the display implementation. This allows concepts such as a 'display' component whose 'user' is actually a separate process.

Download

The download component is responsible for retrieving patch files and their verification signatures and storing them in the filesystem. The only download component currently supplied with the system is a simple anonymous FTP mirroring system that downloads everything it finds in a given directory. However, the interface supports practically any kind of download that can be represented by a standard URL for new means of downloading files, and new modules can be derived which are more selective about which

files are downloaded. For example, a "Solaris-PatchFTP" module could be implemented that knows exactly which files on a Sun FTP site are applicable Solaris patches, and only retrieves those files. The generic FTP module puts more power and responsibility in the hands of the user, since the user has to decide which files have meaning and deal with them appropriately. Because this "sledgehammer" approach is not always desirable, the download mechanism can be easily refined and extended to make Cro-Magnon a more delicate tool.

In order to create a new download component, an implementor needs to provide only two functions: a constructor, which takes a URL and does any required parsing and initialization, and a download method which retrieves the file. The basic FTP module breaks this method up into several submethods, any of which can be individually overridden. This allows for extensions as simple as recursive mirroring of subdirectories or usage of an FTP site which requires a customer login, or as complex as transferring files via HTTP or from a CD-ROM.

Verification

The verification component is responsible for performing security authentication of the trustworthiness of a downloaded file. Two examples are a "Simple" verifier and a MD5 checksum verifier. The simple verifier just trusts all files to be secure. The primary purpose of this module is to serve as a place holder and define the expected API for future authentication schemes, but it also works for a patch site that is implicitly trusted. The MD5 checksum verifier calculates the checksum of the downloaded patch, and compares it against a downloaded checksum before giving approval for the file. The verification interface is generic enough that it can support many other verification schemes, including PGP signature verification.

To implement a new verification scheme, the module requires only two functions: a constructor which does any required initialization, and a verify method, which takes an array of filenames to be checked, and returns an array which is the subset of filenames which have passed verification. This allows the user to pass in any required files, such as the patch file and the MD5 checksum (or PGP signatures), use them as needed to perform the verification, and return the patches which have passed.

Notification

The notification component takes a list of administrator IDs and a list of downloaded patches and notifies those administrators that the patches have been received. The currently supplied notification module does email notification and assumes that the administrator IDs are email addresses. However, since the ID is free-form text, it may represent anything relevant to the appropriate module: for example, if a notification module is used that writes a message to the administrators' workstation consoles, the administrator IDs could be machine names.

Related Work

Although we presume that most large networks use homegrown automated tools to provide many of the functions included in Cro-Magnon, little has been published about such systems. One goal of the Cro-Magnon project is to provide a freely-available alternative to reinventing the wheel at every new site.

One of the most commonly-used such solutions is some variation on one of the widely-available "mirror" packages. These systems mirror various FTP sites to local directories as Cro-Magnon does, but have no built-in capability for patch verification or notification of administrators. In addition, many vendors have convoluted FTP structures that cannot be handled by simply mirroring a directory. Cro-Magnon modules can be written to deal with these sites, but no easy alternative exists with mirror packages.

Another solution that is fairly easily implemented is a combination of cron and the GNU wget package, which can recursively mirror Web and FTP sites. Wget offers many benefits over a simple mirroring strategy including the ability to do wildcard matching on filenames and good handling of slow or unstable connections. However, the cron/wget combination does not automatically support patch verification or administrator notification. Some sort of additional functionality must be added to provide these capabilities. Wget's strengths could be utilized by Cro-Magnon by creating a download module that calls wget.

A more complete home-grown system is the SAGE-AU (System Administrators Guild of Australia) FTP site (ftp://ftp.sage-au.org.au/), which performs local mirroring of system patches. This system was developed (and is only used) in-house by SAGE-AU. SAGE-AU uses a publicly available 'mirror' Perl script to handle the mirroring of patches to a central location. This script is launched multiple times in parallel, one process for each site, to prevent efficiency bottlenecks and to prevent crashing the whole system if a single connection is hung or broken. Currently, a SQL database is created from the output of the mirroring script. Another process compares this database with the registered interests of each user of the system, and email is generated to notify the appropriate users.

Because the purpose of this system is to mirror available patches, rather than directly gather the patches for installation by an end user, there is no need for the SAGE-AU system to authenticate the patches it mirrors. The end user will want to authenticate these patches, just as they would from any other internet site (even if the SAGE-AU administrators are the end users). Even though the purpose of the SAGE-AU system differs from that of Cro-Magnon, Cro-Magnon's robustness and efficiency could be improved by

studying the design of the SAGE-AU system. The most immediately applicable improvement is the parallelization of mirroring sites, but some useful insights might also be gained by studying the mirror script and database at the heart of SAGE-AU.

Though built to serve quite a different purpose than Cro-Magnon, the system monitoring tool called Pulsar [4] uses a somewhat similar architecture to provide extensibility and flexibility. Pulsar, written in Tcl/Tk, periodically monitors the health of various components of a computer system using modules specified in a configuration file and reports problems to the user by means of a configurable "pulse monitor." The monitors that are used, the frequency with which they are activated, and the definition of what constitutes an "alarm" are all configured independently by the user.

This design is very similar to Cro-Magnon's in that it provides a great deal of flexibility through the use of pulse monitors which interact with the scheduler and display using a simple API. Like Cro-Magnon's modules, Pulsar's pulse monitors may easily be extended by end users to provide any degree of functionality desired. It is interesting to see how a similar design can be independently applied to two systems with such dissimilar purposes. Further examination of Pulsar's design is warranted.

Future Enhancements

Cro-Magnon has a long list of desired future enhancements; these include:

- Improve system speed and robustness
  - Split the monolithic config file into several smaller files
  - Determine patch dependencies, so that any other required patches are also downloaded, and their dependencies noted when notifying the administrator
  - Bullet-proof the system as much as possible, to avoid single points of failure
  - Parallelization of site access
  - Deal with the situation in which a vendor revokes a patch that was previously available
- More modules
  - FTP user/password authorization
  - HTTP transfers
  - Copies from another directory on the same network
  - Secure transfers via scp/sftp
  - PGP signature verification
- Display improvements
  - Perl/Tk or Java GUI
  - Web-based interface
  - Retrieval and viewing of patch documentation
- Automated patch application
  - Determination of whether a given patch is already installed
  - Patch chaining – cannot install patch y until patch x is installed
- Automated patch distribution
  - Possibly using rdist, or a 'reverse rdist' type of system

Obtaining the Software

Cro-Magnon is still considered pre-alpha release software, and as such is not yet widely available. Please contact the authors for the current software location.

Cro-Magnon requires Perl and several of its modules in order to run. A full listing of requirements is included in the software distribution. All modules may be downloaded from CPAN at www.cpan.org. Minimal requirements are:

- Perl 5.004 or better
- The Sys::Syslog module (standard with the Perl 5.004 distribution)
- The Digest::MD5 module (for MD5 verification)
- The libnet module package (for anonymous FTP downloads)

Acknowledgments

Dan Farmer was helpful in critiquing our design and ideas about the system. He has had many helpful suggestions, most of which we have not yet had time to implement. His suggestions helped us to stay focused on the useful functionality of the system rather than on bells and whistles.

David Conran was willing to share information about the design and inner workings of the SAGE-AU ftp site, which gave us several good ideas for improving Cro-Magnon with respect to efficiency and speed.

The authors of the Net::FTP module (Graham Barr) and the Digest::MD5 module (Gisle Aas) have saved us a lot of work, and we are forever in their debt.

The Perl Cookbook by O'Reilly and Associates had many solutions to problems we encountered in our implementation, including how to detach from the shell for daemon mode and a concise way to parse config file lines.

Our LISA shepherd, Adam Moskowitz, had many useful suggestions for improving the quality of this paper, as did the anonymous referees who critiqued this paper. We are grateful for their help.

And finally, we would like to thank Evi Nemeth, who taught the graduate course in Systems Administration at the University of Colorado, where all this began, and who encouraged us to submit our work to LISA.

Author Information

Jeremy Bargen recently completed his graduate studies at the University of Colorado, Boulder, with a

MS-CS. He currently works for Raytheon Systems Company in Aurora, Colorado, where he is a Senior Software Engineer in the Mission Management Systems organization. Reach Jeremy via U.S. Mail at Raytheon Systems Company; Bldg. S75, M/S A2707; 16800 East Centretech Parkway; Aurora, CO 80011. Or reach him electronically at jbargen@acm.org.

Seth Taplin is pursuing part-time studies for a MS-CS at the University of Colorado, Boulder. He currently works for CiTR, Inc. in Boulder, Colorado, where he is a Senior Software Engineer providing custom software solutions for a variety of domains. Reach Seth via U.S. Mail at CiTR, Inc; 1200 28th Street; Suite 305; Boulder, CO 80303. Or reach him electronically at staplin@acm.org.

References

[1] Dan Farmer, personal communications, April 1999.

[2] *Perl Cookbook*, O'Reilly and Associates.

[3] CPAN documentation for Perl modules, http://www.cpan.org.

[4] R. A. Finkel, "Pulsar: An Extensible Tool for Monitoring Large Unix Sites," *Software Practice and Experience*, Vol. 27(10), 1163-1176.

[5] David Conran, personal communications, August 1999.

[6] R. L. Rivest, *RFC 1321: The MD5 Message-Digest Algorithm*, April 1992.

[7] M.I.T. PGP Distribution, http://web.mit.edu/network/pgp.html.

A Retrospective on Twelve Years of LISA Proceedings

Eric Anderson – University of California, Berkeley
Dave Patterson – University of California, Berkeley

ABSTRACT

We examine two models for categorizing tasks performed by system administrators. The first model is the traditional task based model. The second model breaks tasks down by the source of the problem. We then look at the historical trends of the last 12 years of LISA proceedings based on these models. Finally, we analyze some of the more important tasks done by system administrators and propose future research in those areas. Our hope is that some of the academic rigor in analyzing research can be brought to systems administration without losing the practicality that makes the research valuable.

Introduction

System administrators don't have a lot of time for introspection of their field. So work is repeated and new administrators, or people trying to do research on system administration, don't know where to start. To provide a starting point, we have examined the last twelve years of LISA proceedings and have categorized the papers in two separate ways. One categorization is by problem causes, and has the advantage that it will apply to any task in system administration. The second categorization is the traditional task breakdown, which shows us where past research has been focused.

In addition to categorizing the papers, we examine the trends in the categorization over the last twelve years. We find that some tasks were solved, and then, due to external changes, needed more work. We also find that some tasks have had a remarkable amount of effort focused on them without a complete solution. We then examine in more detail the more popular areas of research both to gain historical understanding and to consider future directions.

So that others can more easily build on our work, we make the complete set of data including both categorizations, brief summaries, and bibliographic information available on the web from http://now.cs.berkeley.edu/Sysadmin/categorization/.

The next two sections examine the two models, and then the fourth section shows historical trends. The fifth section focuses in on a number of important tasks and examines each in detail. The final section provides a few brief conclusions.

A Model of Tasks

The traditional approach for categorization is to group related papers by the task each targets. We did this for all of the papers. We continued the aggregation process starting with the list of tasks having at least two papers and built a hierarchy of tasks as shown below. The categories are sorted by popularity; the paper count is shown in brackets; ties are broken alphabetically. There were a total of 342 papers, and 64 separate categories.

- Services [75]
  - Backup [28]
  - Mail [20]
  - Printing [11]
  - News [5]
  - NFS [4]
  - Web [3]
  - DNS [2]
  - Database [2]
- Software Installation [57]
  - Application Installation [32]
  - OS Installation [14]
  - User Customization [8]
  - Software Packaging [3]
- Monitoring [44]
  - System Monitoring [14]
  - Resource Accounting [6]
  - Data Display [5]
  - Network Monitoring [5]
  - Benchmarking [4]
  - Configuration Discovery [4]
  - Host Monitoring [4]
  - Performance Tuning [2]
- Management [40]
  - Site Configuration [27]
  - Host Configuration [7]
  - Site Move [4]
  - Fault Tolerance [2]
- Miscellaneous [40]
  - Trouble Tickets [9]
  - Secure Root Access [8]
  - General Tool [6]
  - Security [6]
  - File Synchronization [4]
  - Remote Access [3]
  - File Migration [2]
  - Resource Cleanup [2]

- Management [35]
  - Accounts [23]
  - Documentation [4]
  - Policy [3]
  - User Interaction [3]
  - White Pages [2]
- Networking [19]
  - Network Configuration [9]
  - LAN [4]
  - WAN [4]
  - Host Tables [2]
- Improvement [18]
  - Self Improvement [7]
  - Models [5]
  - Software Design [4]
  - Training Administrators [2]
- one paper on topic [19]

We can see that there are many different types of tasks, but a few subjects are very popular: Backup, Mail, Application Installation, Site Configuration, and Accounts. We can also see that there is a remarkable amount of variability among the various tasks, not including the single-paper topics.

The taxonomy is useful because it helps to describe a skill set necessary for system administrators. We can see which areas system administrators have focused most of their efforts on, examine which areas have been successfully solved, and identify areas needing more work. Since this taxonomy is derived from papers, for completeness, it should be combined with tasks from time surveys [Ande95, Kols92] and interviews.

There are some potential concerns about this categorization. The simplest of which is that there were errors in the classification. There were about 350 papers, so a few errors probably occurred in classification. Furthermore, while the first author worked as a system administrator both at SURAnet, and at Carnegie Mellon University, he has clearly not personally performed all of the tasks described. The program committee may also have affected the papers accepted based on their views of what should be in the conference, or because of a limited selection of available papers. Finally, some papers may be missing because companies consider the information to be proprietary. We believe that the classification is useful, but keeping the weaknesses in mind will help prevent us from drawing incorrect conclusions.

A Model of Problem Sources

A second model based on the source of a problem is shown in Figure 1. The source of the problem is labeled on the edges leading out from the center (the happy state) of the state transition diagram. The edges leading back in to the center represent tasks performed to return the system to a happy state. The model was derived in part from the time surveys, which indicated that administrators spent about a third of their time on each of these tasks.

This model is fairly general and hence is able to cover all types of things done by administrators. Either administrators are trying to improve people (training) or trying to improve machines. If they're trying to improve the machines, it's either because the machines need to do something different (configuration management) or because they need to get back to doing what they used to do (maintenance).

Examination of the Different Categories

Configuration management tasks will remain so long as people change how they want to use the system. Only by freezing how the system is used can we eliminate configuration management tasks. Even a simple appliance like a toaster has a few configuration tasks (plugging it in, adjusting the amount of toasting). The tasks have been simplified by limiting choices; adding choices inherently increases complexity.

Maintenance tasks may be eliminated by building systems that recover from internal faults. If a maintenance task can't be eliminated (for example, purchasing replacement hardware), the goal should be

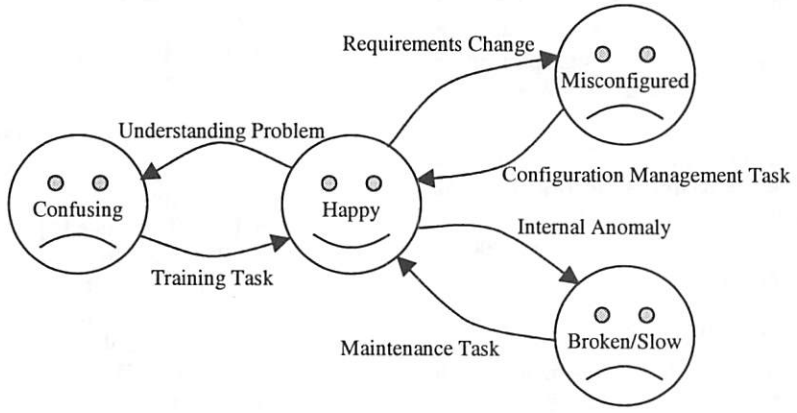


Figure 1: System state transition diagram. Edges out indicate problems that occur making the system less usable. Edges in indicate tasks performed by system administrators to restore the functionality of the system.

to make the task schedulable, rather than forcing an administrator to deal with the problem immediately. Reducing the number of interrupt-style tasks should lead to improving system administrator effectiveness.

Training tasks may be transferable out of the organization and into the schools. Users could be trained in the tools they will be using, and administrators could be trained in system administration. Earlier education would mean people would only have to learn the specifics of a site rather than the general knowledge. Alternately, the various tools that are being used could be improved to reduce the need for training. Researchers in Human Computer Interaction have been looking at this for some time, and have made a number of strides, but more work remains.

Historical Trends of the Conference

Given the two models, we can look at how the papers in the conference have fit into the models over the course of time. This will help us see if things have been changing from previous conferences.

Task Model Trends

Figures 2 and 3 show the papers over the last twelve years categorized by the Task Model. For completeness, we show all of the papers that were shown in the hierarchical categorization.

We can see that some tasks, such as backup, application installation and accounts alternated between very heavy and light years. This probably indicates some amount of duplicated effort in the very heavy years. In some cases (application installation, OS installation), this pattern indicates that good solutions have not been found, and people are still making new, slightly different attempts. In other cases (backup, accounts), it indicates that there was some change in the external world that caused previous solutions to stop working. For example, backup was a task that was successfully solved in the past, but with disk capacity and bandwidth growing faster than tape capacity and bandwidth, it has returned as a problem of dealing with larger scale.

We can see that some tasks, such as printing and trouble tickets, have received a little bit of work fairly steadily. This pattern is probably a good sign, as it means that slow and steady progress is being made without too much duplication of effort.

Mail alternated between the steady work and the heavy work models. Initial work was fairly steady until the explosion of the Internet increased the size of mailing lists, and commercialization resulted in problems with SPAM.

Similarly, some tasks, such as system monitoring and network configuration, see punctuated bursts of activity. This pattern probably indicates that the problem occurred simultaneously due to some external change such as sites scaling up, or new applications. It would be nice if there were some way for different people to coordinate their work as they simultaneously discover new problem areas. This would reduce the amount of duplicated work, and probably also

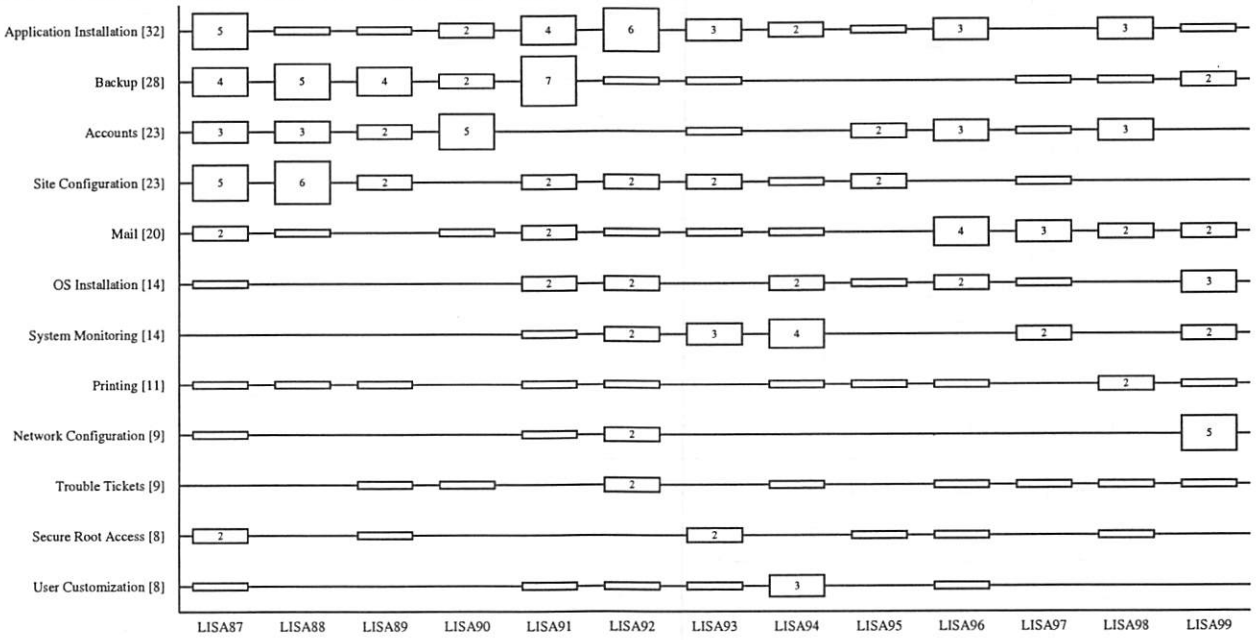


Figure 2: Breakdown of number of papers/conference/category for categories with at least eight total papers. Sorted by popularity of a category, ties broken alphabetically. Height of a box (and the number inside) indicates number of papers. Total number of papers in a category is shown in brackets after the category name. Remainder of categories are shown in Figure 3.

improve the resulting solution as it will deal with the idiosyncrasies of multiple sites.

It is not clear what we can learn from the tasks with fewer papers. In a few cases, we can infer that certain areas did not become problems until fairly recently. Web is an obvious example; configuration discovery, LAN, WAN, and NFS problems also appear to have only become problems recently.

Source Model Trends

Figure 4 shows the papers over the last twelve years categorized by the Source Model.

We can see that the number of training task papers has been remarkably small, and in fact, further examination of the papers in those categories indicates that they are mostly papers on improving the skills of administrators. The one oddity is LISA93, in which a third of the papers were on many different training issues. Some of the training papers cover software design issues for administrators, others suggest how to improve interactions with other administrators, users or managers. A few of the training papers cover how to train new administrators, but surprisingly none of the papers cover training users to take better advantage of software or provide better problem summaries. Training is an area where some work should be done, although it is more difficult to analyze because it involves the variability of people.

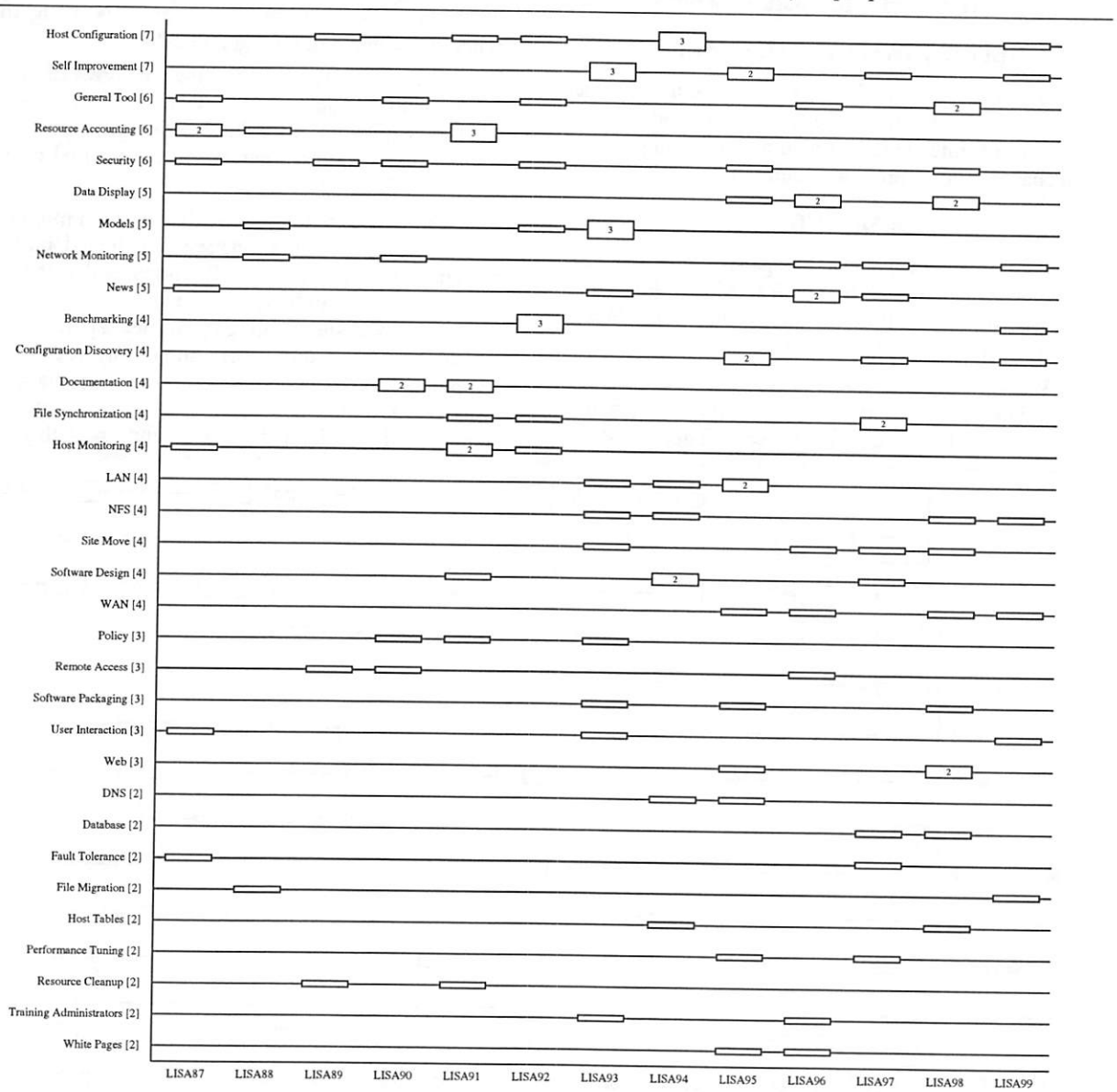


Figure 3: Continuation of Figure 2 for categories with 2-7 papers overall. Sorted by popularity of a category, ties broken alphabetically. This figure is included for completeness, care should be taken in drawing conclusions given the small number of papers.

We can also see that maintenance tasks comprise the second largest fraction of papers. Unfortunately, interrupt-style maintenance tasks contribute greatly to administrator stress. Beyond simply eliminating maintenance tasks by having systems automatically repair themselves, we should strive to convert maintenance tasks to schedulable tasks. If systems were designed to operate in degraded mode, then administrators would not have to respond immediately every time a problem occurred, but could instead work on related tasks at the same time.

Finally, we can see that configuration management tasks are the most prevalent of the papers, which is reasonable and unsurprising given that many tasks eventually require some change in configuration. Configuration tasks generally lead to results which can be more easily described in a paper than results from the other two categories.

Examination of Important Tasks

We now examine the important tasks performed by system administrators in more detail. We summarize the area, examine the research history, and propose directions for future research. Many of the directions would make good papers for future LISA conferences. In the research history, we reference some of the better papers on each topic, so that readers will know where to look for additional information.

Software Installation: OS, Application, Packaging and Customization

Software installation covers the problems of managing software installed on computers. There are four sub-categories of software installation: Operating System (OS) Installation, Application Installation, Software Packaging, and User Customization. Operating system installation deals with the problem of taking the raw machine and putting the operating system on it so it can boot. Application installation is the addition of optional (non-OS) packages to a machine. Software packaging is the step of creating an installable package. User customization happens when users need to change the way the software operates.

Research History

OS installation usually puts files in specific places and has limited support for multiple versions on a single machine. Research into operating system installation has taken a cyclic path. In the very beginning, the OS was installed by either cloning a disk and then putting it in the new machine, or by booting the new machine off some other media (e.g., floppy disk, network) and then copying an image to the local hard drive. Those solutions were then modified to support customization of the resulting installation and easier upgrades [Zwic92, Hide94]. The tools were then scaled to allow fast installation across the entire enterprise [Shad95]. By then large-scale PC OS installation needed to be supported, and the cloning solution [Troc96] reappeared.

Application installation usually puts packages into separate directories, and uses symbolic links to build composite directories, so multiple versions are easily supported, and programs can be beta tested easily before being made generally available. Application

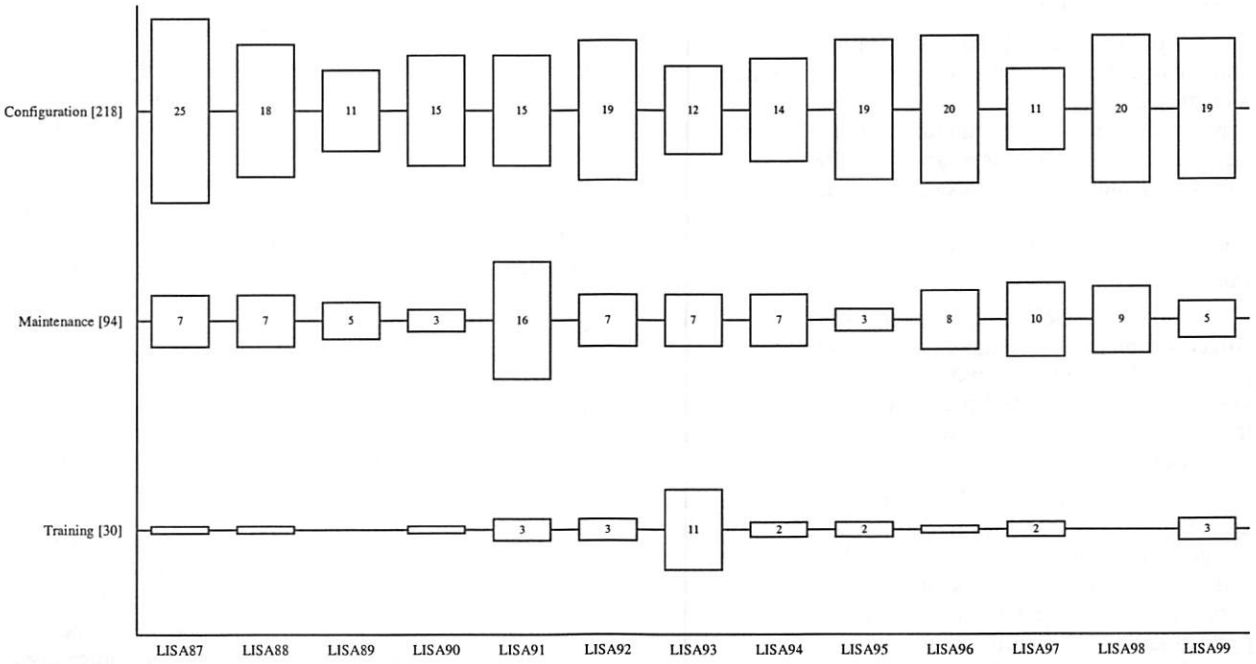


Figure 4: Breakdown of number of papers/conference/category. Sorted by popularity. Height of a box (and the number inside) indicates number of papers. Total number of papers in a category is shown in brackets after the category name.

installation has had many more papers written on it than OS installation, probably because vendors didn't supply tools to install additional applications. The initial solution was to build packages in separate directories and link them into a common directory [Manh90, Coly92]. These tools were then extended to support customization per host [Wong93]. Recently, the caching and linking pieces were untangled and refined into separate tools [Couc96b, Bell96].

Relatively few papers have been written on software packaging, probably because most of the application installation tools use source code trees rather than binary packages. These papers cover the patching of software for different host types, and the subsequent generation of installation packages [Stae98].

The papers on user customization cover two separate areas of customization: Selecting which packages are accessed by a user [Furl96, Will93], and customizing application behavior [Elli92]. The package selection tools started as simple shell scripts that adjusted environment variables to enable packages, and later were refined to work faster and more flexibly. The customization tools have dealt with different aspects of making it easer to control the behavior of programs and have been targeted at beginning users.

An Alternative Breakdown

There have been a remarkable number of papers in this area, many of which seem like slight variations of each other, which makes us wonder if the problem has been broken down poorly. We therefore propose a different breakdown into the following five pieces: Packaging, Selection, Merging, Caching, and End-User Customization.

The distinction between installing applications and the operating systems is probably unnecessary and a historical artifact. Some of the OS installation papers supported some limited number of additional packages, and recent OS installation programs [Hohn99] can install most of the packages available on the net. However, the distinction in functionality that was found in the OS/Application split still remains.

Packaging

Software packaging appears to be a mostly solved problem. There have been a few papers in the LISA conference on it, and the freely available Unix systems have associated packaging tools. Comparing these tools might pave the way to a single multi-platform tool.

Packaging usually binds pathnames into an application. This can limit how packages can be merged later (e.g., two versions both believe they own /usr/lib/package). Some packages allow environment variables to override pathname choices. Exploring the performance and flexibility of the different choices could help improve existing tools.

Selection

Package selection is part of all OS/Application installation tools. The key pieces for a selection tool are the need for per-machine flexibility and the need to support multiple collections. Both programmatic and GUI interfaces should be supported so that the tool is both easier to use and scriptable. The selection tool could then be integrated into some of the existing tools as a uniform front-end.

Merging

Merging packages remains a hard outstanding problem. Many tools just ignore the problem. A few have a configuration file to specify which package overrides another when conflicts occur. Merging is most difficult when packages are inter-related, as is the case with Emacs, Perl and Tcl with their various separate extensions; Tex/LaTeX; X windows with various applications that add fonts and include files; and shared library packages.

One unsatisfying solution is to pre-merge packages during packaging so that there are no inter-relations between packages. A modular solution would need to handle merging of files, for example generating the top level Emacs info file, or the X windows font directory files. Some programs include search paths, which might make the merging easier to handle, others require the execution of a program in the final merged directory.

If multiple versions need to be supported simultaneously, there is a more substantial problem. Supporting the cross product of all possibilities is not practical. However, there is no clear easy solution. Quite a bit of thought will be needed to find an adequate solution.

Caching

Caching to the local disk is beneficial for both performance and for isolating clients from server failures. Caching is a semi-solved problem. Some file systems cache onto local disk to improve performance (e.g., AFS, CacheFS, Coda). In general, caching merely requires mounting the global repository somewhere different and creating symlinks or copies as appropriate. There have been tools written to do just this [Couc96b, Bell96], and many of the general software installation tools have included support for caching [Wong93]. Making the caching fully automatic and fine grained will probably require some amount of OS integration.

End User Customization

End user customization has only been slightly examined. A few tools help users dynamically select the packages they want to use [Furl96]; most have fixed the choice on a per-machine basis. One old paper looked at how users customized their environment [Will93]. It would be nice for this area to be resurrected for research. Programs are becoming increasingly complex, especially as they add GUI interfaces, but the ease of customizing the programs has not kept up. Work in this area would require a large amount of interviewing users to determine what they would like to customize.

Backup

Backup addresses four separate, but related problems: User Error, Independent Media Failure, Correlated Media Failure (e.g., Site Failure, Software Error), and Long Term Storage. All the solutions are based on some type of redundant copy, but the particulars of each are different. Damage due to user error can be reduced by online filesystem snapshots. Independent media failure can be remedied by techniques like RAID. Correlated media failure requires use of additional uncorrelated media (e.g., Off-site tape, remote duplicates with different software). Finally long term storage requires very stable media, and an easily read format. Consider how many people can still read data written on punchcards, or even 9-track tape. Most of the focus in backup has been on independent media failure, usually by creating copies on tape, although people have looked at the other issues.

Research History

Research on backup has passed through many stages. The first was correctness: Does the right data get written? [Zwic91b] Are backups happening regularly and on schedule? [Metz92] Do restores work? Having achieved correctness, research turned to scaling backup solutions to the enterprise. The solution was staging disks so that backups could stream to tape [Silv93]. Having solved the correctness and scalability problems, research on backup paused. But then the onward march of technology reintroduced scalability as a problem. Disk bandwidth and capacity are starting to outstrip tape bandwidth and capacity leading to solutions requiring multiplexing of disks and tapes [Pres98].

Future Directions

Restores seem to be a somewhat overlooked part of the backup problem. Most backup papers deal in great detail with formats of dump tapes, scheduling of backups, streaming to tape. However, they usually only write a few paragraphs on the subject of restores, often ignoring the time taken to restore data. The whole purpose of backup is so that when something goes wrong, restores can happen! We would like a discussion of restore difficulty and measurements of restore performance in future papers. When something fails, there is a cost in lost productivity in addition to the direct cost of performing the repair.

Examining technology trends and technology options would help identify future backup challenges before they occur. The technology involved has reasonably predictable future performance in terms of bandwidth, latency, and capacity. Somewhat weaker predictions can be made about the growth in the storage needs of users. Given this information, a prediction can be made about the required ratio of hardware in the future. In addition, alternatives to tape backup such as high capacity disks and writable cds/dvds may become viable in the future. One advantage of random access media is that data can be directly accessed off the backup media to speed up recovery.

Backup by copying to remote sites is very different from traditional approaches. A few companies are dealing with the possibility of a site failure by performing on-line mirroring to a remote site over a fiber connection. It may be possible to decrease the required bandwidth by lowering the frequency of the updates, so that this approach is practical for people unable to purchase a dedicated fiber.

Backups also present special security concerns. A backup is typically an unprotected copy of data. If anyone can get access to backup tapes, they can read critical data. How can encryption be used to solve the security problem? Will encryption enable safe web backup systems?

Another interesting question is how to handle backup for long-term storage. Some industries have legal requirements to retain documents for a long (indefinite) time. There are two related problems. First, media needs to be found which is stable enough to last a long time. Second, it seems wise to rely on conversion to a common format because it is never clear what software will still work in 20-50 years. How can these two concerns be integrated into a backup solution?

Configuration: Site, Host, Network, Site Move

Configuration tasks are modification to the setup of hardware and software so that the environment matches the requirements of a particular organization. These tasks can range from simply installing the appropriate exports and resolv.conf files to complicated tasks like migration from an MVS platform to a UNIX one.

Research History

The first few LISA conferences included many papers which summarized their site's configuration. Research then forked in two directions. Some papers looked at how to store and extract configuration information from a central repository, either using available tools such as SQL [Fink89], or by designing their own language [Roui94a]. Other papers looked at using a level of indirection to make configuration changes transparent to users [Detk91].

The great growth spurt in the computer industry lead to complete site moves, either as part of a merger, separation, or just to handle growth [Schi93]. Similarly, the great amount of research in this area led some people to examine the question, "What properties of site design make it easier to administer?" [Trau98]. Recently, a mobile user base caused dynamic network re-configuration to become a problem [Vali99].

Categorization Commentary

This is probably the weakest categorization. The original intent was that host configuration would cover host issues, network configuration would cover network issues, and site configuration would cover global site issues. However, the line between host and site is

at best blurry. We therefore believe that someone should re-examine the papers in these areas, and see if they can find a better categorization.

Future Directions

The key to host configuration seems to be having a central repository of information that is then pushed or pulled by hosts. Most of the papers did some variant of this. Two areas remain to be refined: First, someone should analyze exactly what information should be in the central repository, and how it can be converted to the many different types of hosts in use. Second, someone should write a tool to automatically create the repositories so that the start-up cost to using a configuration tool is lower.

Site configuration tools vary widely, probably because of the different requirements at each site (e.g., a wall street trading firm vs. a research lab). [Evar97] surveyed the current practices, and [Trau98] studied the best practices for certain environments. Combining these two directions by identifying the best practices based on the requirements of a site would help all sites do a better job of configuration.

Network configuration is a fairly recent topic, so proposing directions by analyzing the papers is risky. However, we can still look at analogies to previous work. First, we want to build abstract descriptions of the system. Second, the models should be customizable; early configuration tools didn't support much customization, so later ones had to add it. Third, a survey paper, analogous to [Evar97] would help identify the problems in network configuration research.

Accounts

Managing user accounts at first seems very simple. But further examination indicates that there are additional subtleties because an account identifies users, and therefore has lots of associated real world meaning. Therefore, authentication, rapid account creation, and managing the associated user information become important.

Research History

Accounts research started with the goal of simplifying the account creation process. Scripts were designed that automated the steps of accumulating the appropriate information about users, adding entries to password files, creating user directories, and copying user files [Curr90]. Because the scripts were site-specific, they were able to do better error checking. Once creating accounts became easy, accounts research paused until enough people needed accounts that scalability became a concern. Sites with thousands of accounts, usually schools, needed to create lots of accounts quickly because of high turnover in the user population. Their solutions tended to have some sort of central repository storing account information (often an admissions' database), with complementary daemons on client nodes to extract the needed parts of the database [Spen96]. Some of the recent papers considered auxiliary details such as limiting accounts to certain hosts, account expiration, and delegating authority to create accounts [Arno98].

Future Research Opportunities

Surveying account creation practices would help identify why no tool has evolved as superior despite many papers on this subject. We believe this is because of unrecognized differences in the requirements at each site. With all the requirements explicitly described, it should be possible to build a universal tool.

A related topic is the examination of specific issues related to account creation. For example, many of the papers ignored the question of how to limit accounts to specific machines. Is a simple grouping as was done for host configuration sufficient, or is some sort of export/import setup needed? Sharing accounts across administrative boundaries within an organization will make this problem even more difficult.

Another specific issue is delegation of account creation. The one tool to do this [Arno98] assumed all the employees were trusted to enter correct account information. Clearly this solution will not work at all sites. There may be synergy with the secure root access papers that looked at delegation.

Mail

Electronic mail has been one of the driving applications on the Internet since its inception. This makes it unsurprising that it ranks extremely high on the list of applications. It is the highest of the applications that are used by end-users on a regular basis. There is a vast amount of email, traveling around the world-wide network, leading to a lot of effort in interoperability and scalability.

Research History

Very early research in mail targeted interoperability between the wide variety of independently developed mail systems. This research and the reduction in variety over time, combined with SMTP as a standard mail interchange protocol, solved the interoperability problem. Research then turned to flexible delivery and automating mailing lists [Chap92]. There was then a brief pause in the research. However, as the Internet continued to grow, research on scaling delivery of mail both locally and in mailing lists [Kols97] was needed. At the same time, commercialization caused SPAM to become a problem [Hark97].

Future Research

The biggest remaining problem is dealing with SPAM. The correct solution is probably dependent on trading off difficulty in being reached legitimately with protection from SPAM. Some possible approaches are: acceptance lists with passwords, a list of abusers that are automatically ignored (this is being done), a pattern matcher for common SPAM forms, and receive-only/send-only addresses. Finding a good solution will be challenging.

Scalability and security still need some work. Scalability of mail transport and mail delivery may be possible by gluing together current tools into a clustered solution. Both problems partition easily. Handling more types of security threats also remains open; [Bent99] has done some initial work securing MTA ⇔ MTA transfers.

Monitoring: System, Network, Host, Data Display

Monitoring solutions help administrators figure out what is happening in the environment. There are problems of system, network and host monitoring, and the associated problem of data display. Monitoring solutions tend to have two variants: instantaneous and long term.

Research History

Research in monitoring has progressed along a number of axes. First, there has been work in monitoring specific sources from file and directory state [Rich91] to OC3 links [Apis96]. Simultaneously, generic monitoring infrastructure [Hard92, Ande97a] has been developed. Finally, as the amount of data available has increased, some work on data display has been done [Oeti98a].

Categorization Commentary

The categorization here was by the type of thing being monitored (host, network system). Perhaps a better classification would be by the axes described in the research history.

Future Directions

There has been a lot of work on gathering data from specific sources, but in most cases, the overhead for gathering data has been high, so the interval is usually set in minutes. Reducing this overhead is important for allowing finer grain monitoring [Ande97b]. In addition, we would like to vary the gathering interval so that the overhead of fine-grain gathering is only incurred when the data would be used. In addition to just gathering the data, having a standard form for storing the data efficiently would be very useful. Combining these two issues should lead to a nice universal tool with pluggable gathering modules.

Data analysis and data reduction have not received nearly the attention they deserve. The data collection techniques are only useful if the data can be used to identify problems. But beyond averaging time-series data, very little automated analysis has been done. An examination of methods for automated analysis, for example, looking at machine learning techniques, could prove fruitful.

Data visualization has started to get some examination in the system administration field. There is a vast amount of literature on various forms of visualization in the scientific computing field. We believe that a survey of existing techniques would lead to tools that allow visualization in system administration to be both more effective and more scalable.

Printing

Printing covers the problems of getting print jobs from users to printers, allowing users to select printers, and getting errors and acknowledgements from printers to users.

Research History

Early research in printing merged together the various printing systems that had evolved [Flet92b]. Once the printing systems were interoperable, printing research turned to improving the resulting systems, making them easier to debug, configure, and extend [Powe95]. As sites continued to grow, scaling the printing system became a concern, and recent papers have looked into what happens when there are thousands of printers [Wood98].

Future Directions

Printing research seems to be in fairly good shape. Scaling print systems is still not completely done, debugging problems and selecting the right printer is still challenging. Perhaps printer selection could be done by property (e.g., color, two sided). Finally, the path for getting information from printers back to users has not been well examined. A notification tool to tell users the printer's status, such as print job finished or out of paper, would be useful. The notification tool might also help in debugging printing problems.

Trouble Tickets

Trouble ticket tools simplify the job of accepting a problem report, assigning the problem report to an administrator, fixing the problem, and closing the problem's ticket. Trouble ticket systems usually have a few methods for getting requests into the system (e-mail, phone, GUI), and provide tools for querying and adjusting the requests once they are in the system.

Research History

Trouble ticket systems began as email-only submission tools with a centralized queue for requests [Galy90]. Later, the systems were extended so that users could query the status, and tickets could be assigned to particular administrators [Kobl92]. The systems were improved to support multiple submission methods such as phone [Scot97] and GUI, and to support multiple request queues [Ruef96].

Future Directions

There seems to be a fair amount of overlap in the research on trouble tickets. Many of the tools were created from scratch, only occasionally building on the previous research. Examining the existing tools should identify the different requirements that have led to all these systems and to a more general tool.

A second direction to extend trouble ticket systems would be to build in a knowledge of the request handling process. [Limo99] examines the process of handling problem reports, but doesn't propose tools. A trouble ticket system supporting the process would be quite valuable.

Secure Root Access

Secure root access is the general problem of providing temporary privileges to a partially trusted user. Many actions need to be taken as root, and giving out the root password is clearly a poor decision. The questions then are how to give out privileges, how to track their use, and how to retain some amount of security.

Research History

Research in secure root access has gone down two separate paths. One path has been to examine how to provide secure access to commands within a host. This has gone through many iterations, slowly adding in more complex checking of programs and arguments [Mill99, Hill96]. The other has been to provide secure access remotely [Ramm95].

Future Directions

The unfortunate effect of having the two separate paths of research is that neither handles all the problems easily. The remote tools are more flexible, but harder to configure, and don't support logging well. The local tools have a more natural interface, but don't have as much power to provide partial access. Combining these two paths of research should lead to a more powerful and flexible tool.

A second direction to consider is toward providing finer-grain access control. [Gold96] did this by securely intercepting system calls. Further work could lead to having something like capabilities in the OS, allowing very precise control over the access granted to partially-privileged users.

Conclusions and Analysis

We have categorized all of the papers in the LISA conference according to two separate models. We have made the categorization available so that others can examine our choices, correct mistakes, or provide better categorizations. Hopefully this paper will encourage people to think differently about the field and problems that it presents, and as a result build better tools and processes.

We would like to see other people examine some of the other conferences that may publish relevant papers. The USENIX general conference, SIGCOMM, and SANS are a few places to start looking. There is likely to be some useful information present in those conferences which was not covered in this paper.

We have examined the historical trends of the LISA conference according to the two models. This has helped us see that some areas are under served, and some are probably over-served. We can also see the bursty nature of research in system administration (probably because the same problem occurs to everyone at the same time). As a result we recommend that a central clearinghouse of problems be created to facilitate collaboration and improve the resulting tools.

Finally we examined some of the important task areas. Based on our analysis, we have proposed a number of papers to be written. We believe that this sort of analysis should be performed every few years. The Database community gets together and decides which areas of research were successful, and which require more work [Silb91, Silb96]. Their reports have helped their community show their results and focus their efforts. Hopefully this analysis of system administration will help do the same for ours.

Acknowledgements

We would like to thank Evi Nemeth, Kim Keeton, Drew Roselli, Aaron Brown, and David Oppenheimer, and the anonymous reviewers for their comments on the paper. Their comments have improved both the ideas and the readability of the paper immensely. This work was supported by DARPA under grant DABT63-96-C-0056.

References

The entire database of categorized papers is available from http://now.cs.berkeley.edu/Sysadmin/categorization/.

[Ande95] Eric Anderson. "Results of the 1995 SANS Survey" *;login:*, October 1995, Vol20, No. 5, http://now.cs.berkeley.edu/Sysadmin/SANS95-Survey/index.html; Found weak correlation between number of machines and number of admins; many important tasks.

[Ande97a] Eric Anderson and Dave Patterson, "Extensible, Scalable Monitoring for Clusters of Computers," *Proceedings of the Eleventh Systems Administration Conference* (LISA '97), San Diego, California pp 9-16; http://now.cs.berkeley.edu/Sysadmin/esm/intro.html; Tool for monitoring and displaying cluster statistics.

[Ande97b] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandervoorde, Carl A. Waldspurger, and William E. Weihl. "Continuous Profiling: Where Have All the Cycles Gone?" *16th ACM Symposium on Operating Systems Principles,* Saint-Malo, France, http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/abstracts/src-tn-1997-016.html.

[Apis96] Joel Apisdorf, k claffy, Kevin Thompson, and Rick Wilder, "OC3MON: Flexible, Affordable, High Performance Statistics Collection," *Proceedings of the Tenth Systems Administration Conference* (LISA '96), Chicago, Illinois, pp 97-112, http://www.nlanr.net/NA/Oc3mon/; HW & SW for monitoring and analyzing traffic on an OC3 link.

[Arno98] Bob Arnold, "Accountworks: Users Create Accounts on SQL, Notes, NT, and UNIX," *Proceedings of the Twelfth Systems Administration Conference* (LISA '98), Boston, Massachusetts, pp 49-61.

[Bell96] John D. Bell. "A Simple Caching File System for Application Serving," *Proceedings of the Tenth Systems Administration Conference* (LISA '96), Chicago, Illinois, pp 171-179; Automatic caching of applications from a remote server to local disk.

[Bent99] Damien Bentley, Greg Rose, and Tara Whalen. "ssmail: Opportunistic Encryption in sendmail," *Proceedings of the Thirteenth Systems Administration Conference* (LISA '99), Seattle, Washington.

[Chap92] D. Brent Chapman. "Majordomo: How I Manage 17 Mailing Lists Without Answering '-request' Mail," *Proceedings of the Sixth Systems Administration Conference* (LISA '92), Long Beach, California, pp 135-143, ftp://ftp.greatcircle.com/pub/majordomo.tar.Z .

[Coly92] Wallace Colyer and Walter Wong, "Depot: A Tool for Managing Software Environments," *Proceedings of the Sixth Systems Administration Conference* (LISA '92), Long Beach, California, pp 153-162, ftp://export.acs.cmu.edu/pub/depot/; Build merged tree by copy/link from packages; conflict resolution by package preferences.

[Couc96b] Alva L. Couch, "SLINK: Simple, Effective Filesystem Maintenance Abstractions for Community-Based Administration," *Proceedings of the Tenth Systems Administration Conference* (LISA '96), Chicago, Illinois, pp 205-212, ftp://ftp.cs.tufts.edu/pub/slink; Flexible sym-linking/copying for merging software repositories.

[Curr90] David A. Curry, Samuel D. Kimery, Kent C. De La Croix, and Jeffrey R. Schwab, "ACMAINT: An Account Creation and Maintenance System for Distributed UNIX Systems," *Proceedings of the Fourth Large Installation Systems Administrator's Conference* (LISA '90), Colorado, pp 1-9.

[Detk91] John F. Detke. "Host Aliases and Symbolic Links -or- How to Hide the Servers' Real Name," *Proceedings of the Fifth Large Installation Systems Administration Conference,* (LISA '91), San Diego, pp 249-252; Use host aliases & symbolic links to allow mount points & servers of exported FS to move w/o needing client changes.

[Elli92] Richard Elling and Matthew Long. "usersetup: A System for Custom Configuration of User Environments, or Helping Users Help Themselves," *Proceedings of the Sixth Systems Administration Conference* (LISA '92), Long Beach, California, pp 215-223, ftp://ftp.eng.auburn.edu/; Extension to Modules [Furl91] system, menu driven script to select applications & configure apps.

[Evar97] Rémy Evard. "An Analysis of UNIX System Configuration," *Proceedings of the Eleventh Systems Administration Conference* (LISA '97), San Diego, California, pp 179-193; Examination of current configuration practices at nine different sites.

[Fink89] Raphael Finkel and Brian Sturgill. "Tools for System Administration in a Heterogeneous Environment," *Proceedings of the Workshop on Large Installation Systems Administration III* (LISA '89), Austin, Texas, pp 15-29; Relational structure stores host, file information. Tables can be generated at runtime. Schema describes relation structure & constraints. Query language queries & executes.

[Flet92b] Mark Fletcher, "nlp: A Network Printing Tool," *Proceedings of the Sixth Systems Administration Conference* (LISA '92), Long Beach, California, pp 245-256; Centralized print server database, uses lpd protocol to transfer files.

[Furl96] John L. Furlani and Peter W. Osel, "Abstract Yourself With Modules," *Proceedings of the Tenth Systems Administration Conference* (LISA '96), Chicago, Illinois, pp 193-203, http://www.modules.org/; Per-user flexible configuration of accessible packages.

[Galy90] Tinsley Galyean, Trent Hein, and Evi Nemeth, "Trouble-MH: A Work-Queue Management Package for a >3 Ring Circus," *Proceedings of the Fourth Large Installation Systems Administrator's Conference* (LISA '90), Colorado, pp 93-95.

[Gold96] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer, "A secure environment for untrusted helper applications: confining the wily hacker," *Sixth USENIX Security Symposium, Focusing on Applications of Cryptography,* San Jose, California, http://www.cs.berkeley.edu/~daw/janus/ .

[Hard92] Darren R. Hardy and Herb M. Morreale, "buzzerd: Automated Systems Monitoring with Notification in a Network Environment," *Proceedings of the Sixth Systems Administration Conference* (LISA '92), Long Beach, California, pp 203-210; Central monitoring server, remote monitoring daemons, paging on problems, users can put in notification, filtering, and escalation.

[Hark97] Robert Harker, "Selectively Rejecting SPAM Using Sendmail," *Proceedings of the Eleventh Systems Administration Conference* (LISA '97), San Diego, California, pp 205-220, http://www.harker.com/sendmail/anti-spam; Configuring sendmail to reject spam messages.

[Hide94] Imazu Hideyo. "OMNICONF – Making OS Upgrads and Disk Crash Recovery Easier," *Proceedings of the Eighth Systems Administration Conference* (LISA '94), San Diego, California, pp 27-31; Calculate a delta between two configurations, store the delta, apply it later.

[Hill96] Brian C. Hill, "Priv: Secure and Flexible Privileged Access Dissemination," *Proceedings of the Tenth Systems Administration Conference* (LISA '96), Chicago, Illinois, pp 1-8, ftp://ftp.

ucdavis.edu/pub/unix/priv.tar.gz; Secure ability to run programs as root with flexible command checking.

[Hohn99] Dirk Hohndel, "Automated installation of Linux systems using YaST," *Proceedings of the Thirteenth Systems Administration Conference* (LISA '99), Seattle, Washington.

[Kobl92] David Koblas, "PITS: A Request Management System," *Proceedings of the Sixth Systems Administration Conference* (LISA '92), Long Beach, California, pp 197-202; Users can query database of open tickets, centralized assignment of new tickets, request editing tool.

[Kols92] Rob Kolstad, "1992 LISA Time Expenditure Survey," *;login:*; Administrator time spread over many tasks.

[Kols97] Rob Kolstad, "Tuning Sendmail for Large Mailing Lists" *Proceedings of the Eleventh Systems Administration Conference* (LISA '97), San Diego, California, pp 195-203; Configuring sendmail to increase performance.

[Limo99] Thomas A. Limoncelli. "Deconstructing User Requests and the 9-Step Model," *Proceedings of the Thirteenth Systems Administration Conference* (LISA

[Manh90] Kenneth Manheimer, Barry A. Warsaw, Stephen N. Clark, and Walter Rowe, "The Depot: A Framework for Sharing Software Installation Across Organizational and UNIX Platform Boundaries," *Proceedings of the Fourth Large Installation Systems Administrator's Conference* (LISA '90), Colorado pp 37-46.

[Metz92] Melissa Metz and Howie Kaye, "DeeJay – The Dump Jockey: A Heterogeneous Network Backup System," *Proceedings of the Sixth Systems Administration Conference* (LISA '92), Long Beach, California, pp 115-125, ftp://ftp.cc.columbia.edu/.

[Mill99] Todd Miller, Dave Hieb, Jeff Nieusma, Garth Snyder, et al., "Sudo: a utility to allow restricted root access," http://www.courtesan.com/sudo/ .

[Oeti98a] Tobias Oetiker, "MRTG – The Multi Router Traffic Grapher," *Proceedings of the Twelfth Systems Administration Conference* (LISA '98), Boston, Massachusetts, pp 141-147, http://ee-staff.ethz.ch/~oetiker/webtools/mrtg/3.0/ .

[Powe95] Patrick Powell and Justin Mason, "LPRng – An Enhanced Printer Spooler System," *Proceedings of the Ninth Systems Administration Conference* (LISA

[Pres98] W. Curtis Preston, "Using Gigabit Ethernet to Backup Six Terabytes," *Proceedings of the Twelfth Systems Administration Conference* (LISA '98), Boston, Massachusetts, pp 87-95.

[Ramm95] Karl Ramm and Michael Grubb, "Exu – A System for Secure Delegation of Authority on an Insecure Network," *Proceedings of the Ninth Systems Administration Conference* (LISA '95), Monterey, California, pp 89-93, ftp://ftp.duke.edu/pub/exu; A tool for providing fine-grain root access via authenticated, privileged scripts.

[Rich91] Kenneth Rich and Scott Leadley, "hobgoblin: A File and Directory Auditor," *Proceedings of the Fifth Large Installation Systems Administration Conference* (LISA '91), San Diego, pp 199-207, ftp://cc.rochester.edu/ftp/pub/ucc-src/hobgoblin; list of files/dirs + attributes => model. Checks for correctness, autogenerated from tar or ls listings.

[Roui94a] John P. Rouillard and Richard B. Martin, "Config: A Mechanism for Installing and Tracking System Configurations," *Proceedings of the Eighth Systems Administration Conference* (LISA '94), San Diego, California, pp 9-17, ftp://ftp.cs.umb.edu/pub/bblisa/talks/config/config.tar.Z; Update target machines using rdist + make with master repository in CVS, look for changed files with tripwire.

[Ruef96] Craig Ruefenacht, "RUST: Managing Problem Reports and To-Do Lists," *Proceedings of the Tenth Systems Administration Conference* (LISA '96), Chicago, Illinois, pp 81-89, ftp://ftp.cs.utah.edu/pub/rust; Manages trouble ticket reports via e-mail.

[Schi93] John Schimmel, "A Case Study on Moves and Mergers," *Proceedings of the Seventh Systems Administration Conference* (LISA '93), Monterey, California, pp 93-98, How the merger of SGI & Mips was handled, physical move & computer configuration issues.

[Scot97] Peter Scott, "Automating 24x7 Support Response To Telephone Requests," *Proceedings of the Eleventh Systems Administration Conference* (LISA '97), San Diego, California, pp 27-35, Phone system for receiving problem reports and paging people.

[Shad95] Michael E. Shaddock, Michael C. Mitchell, and Helen E. Harrison, "How to Upgrade 1500 Workstations on Saturday, and Still Have Time to Mow the Yard on Sunday," *Proceedings of the Ninth Systems Administration Conference* (LISA '95), Monterey, California, pp 59-65; Tools & processes used to quickly upgrade an entire site.

[Silb91] Avi Silberschatz, Michael Stonebraker, and Jeffrey D. Ullman, "Database Systems: Achievements and Opportunities," *Communications of the ACM*, 34(10), pp 110-120.

[Silb96] Avi Silberschatz, Michael Stonebraker, and Jeffrey D. Ullman, "Database Research: Achievements and Opportunities into the 21st Century," Stanford Technical Report, http://elib.stanford.edu/ as CS-TR-96-1563.

[Silv93] James da Silva and Ólafur Guômundsson, "The Amanda Network Backup Manager," *Proceedings of the Seventh Systems Administration Conference* (LISA '93), Monterey, California, pp 171-182, ftp://ftp.cs.umd.edu/pub/amanda; Network backup by staging to a holding disk & streaming to tape, flexible scheduling.

[Spen96] Henry Spencer, "Shuse: Multi-Host Account Administration," *Proceedings of the Tenth Systems Administration Conference* (LISA '96), Chicago, Illinois, pp 25-32, Centralized account management for rapid account creation.

[Stae98] Carl Staelin, "mkpkg: A software packaging tool," *Proceedings of the Twelfth Systems Administration Conference* (LISA '98), Boston, Massachusetts, pp 243-252, http://www.hpl.hp.com/personal/Carl_Staelin/mkpkg; Automatically generate manifest & install scripts; backend makes package.

[Trau98] Steve Traugott and Joel Huddleston, "Bootstrapping an Infrastructure," *Proceedings of the Twelfth Systems Administration Conference* (LISA '98), Boston, Massachusetts, pp 181-196.

[Troc96] Jim Trocki, "PC Administration Tools: Using Linux to Manage Personal Computers," *Proceedings of the Tenth Systems Administration Conference* (LISA '96), Chicago, Illinois, pp 187-192; Installation of DOS/Windows using Linux boot disk.

[Vali99] Peter Valian and Todd K. Watson, "NetReg: An Automated DHCP Network Registration System," *Proceedings of the Thirteenth Systems Administration Conference* (LISA '99), Seattle, Washington.

[Will93] Craig E. Wills, Kirstin Cadwell, and William Marrs, "Customization in a UNIX Computing Environment," *Proceedings of the Seventh Systems Administration Conference* (LISA '93), Monterey, California, pp 43-49; Study of how users customize their environment (copied from friends, then changed).

[Wong93] Walter C. Wong, "Local Disk Depot – Customizing the Software Environment," *Proceedings of the Seventh Systems Administration Conference* (LISA How to cache packages onto the local disk in the depot [Coly92].

[Wood98] Ben Woodard, "Building An Enterprise Printing System," *Proceedings of the Twelfth Systems Administration Conference* (LISA '98), Boston, Massachusetts, pp 219-228, http://pasta.penguincomputing.com/pub/prtools .

[Zwic91b] Elizabeth D. Zwicky, "Torture-testing Backup and Archive Programs: Things You Ought to Know But Probably Would Rather Not," *Proceedings of the Fifth Large Installation Systems Administration Conference* (LISA '91), San Diego, pp 181-189; Static & dynamic tests for many backup programs (dump, cpio, tar, etc.) shows many problems.

[Zwic92] Elizabeth D. Zwicky, "Typecast: Beyond Cloned Hosts," *Proceedings of the Sixth Systems Administration Conference* (LISA '92), Long Beach, California, pp 73-78, ftp://ftp.erg.sri.com/pub/packages/typecast .

Managing Security in Dynamic Networks

Alexander V. Konstantinou and Yechiam Yemini – Columbia University
Sandeep Bhatt and S. Rajagopalan – Telcordia Technologies (formerly Bellcore)

ABSTRACT

This paper describes our initial steps towards self-configuring mechanisms for automating high-level security and service policies in dynamic networks. We build on the NESTOR system developed at Columbia University for instrumenting and monitoring constraints on network elements and services such as DHCP, DNS zones, host-based access controls, firewalls, and VLAN switches.

Current paradigms for configuration management require that changes be propagated either manually or via low-level scripts suited to static networks. Our longer-term goal is to provide fully automated techniques which work for dynamic networks in which changes are frequent and often unanticipated. Automated approaches, such as ours, are the only viable solution for global and dynamic networks and services. In this paper, we focus on one specific scenario to illustrate our ideas: providing transparent and secure access to selected services from a mobile laptop. The challenge is that reconfiguration must satisfy the security policies of two independent corporate networks.

Introduction

As the technologies to deploy new internet services have progressed rapidly, the tools to manage them have lagged behind. This is especially true of security management. Sophisticated services are often vulnerable to unanticipated and sophisticated attacks. With inadequate management tools, the tendency is to deploy new services in a restricted manner that limits their usefulness.

Existing tools for static security are inadequate to meet the current demands of user mobility and diversity. These tools require frequent, expensive, and error-prone reconfigurations that may make legitimate access cumbersome and time consuming. This happens because the primary first-generation technique for reconfiguration – low-level scripting – cannot handle rapid change easily. There are no tools to verify the correctness or composability of scripts, properties critical for security.

As a result, unpredictable security gaps can appear during changeovers. In a dynamic network with frequent upgrades, this uncertainty becomes untenable and leads to over or under-management of resources. The task of systems administration is increasingly human-intensive and administrators often must make decisions with little or no basis to justify their choice. The need for automation of configuration management is immediate. History also tells us that any security mechanisms that obstruct the legitimate use of services become unpopular and will eventually be bypassed. Balancing the demand of users for new services with the security vulnerabilities that the new services cause is an important and challenging problem.

The long-term goal of our project is a management platform that automates both the management of security and service availability. This paper describes our approach and initial steps towards this goal. We demonstrate our solution with a simple scenario in which a user moves from one network to another while expecting transparent access to services. This example illustrates the complexities of reconfiguring networks which are independently administered. It should be noted that while we describe our solution for one specific example in this paper, our approach is more general and geared towards the larger problems of service and security management. The scenario in this paper is a vehicle to illustrate our ideas; indeed, we do not address every conceivable aspect of the scenario.

We describe how we use the NESTOR configuration management system prototyped at Columbia University to manage security requirements. We also discuss our plans for further work towards self-configuring network systems that maintain high-level security and service policies dynamically.

The Scenario

Jane Consultant, who is employed by Corporation A, is visiting a client in Corporation B. During her meeting, Jane realizes that she needs to access files in her home directory which have not been copied onto her laptop. When plugged into her home network in A, Jane simply clicks an icon on her desktop to access her files. What are her choices while plugged into B's network? She can establish a slow but potentially insecure modem connection to Corporation A (over a wireless connection for example, or perhaps the phone call is routed over the Internet). Alternatively, she can plug her laptop into an ethernet port within Corporation B; assuming she gets connected at all, it will likely be a window-less connection to B because Corporation A may not open X

services in its network to hosts outside. Neither method offers access comparable to what Jane would get within her home network.

What makes the problem more challenging is that the two networks are separately administered, with independent security policies. For example, Corporation A might filter certain services when the user is plugged into a remote network. Corporation B might require that guest machines not be able to send or receive traffic directly from any machine within B's network, and that guest machines may only access remote VPN nodes. If there is a way to provide access without violating either company's policy, we would like all necessary reconfigurations to be automatic and not require manual intervention. Of course, if there is no way to provide access without violating one security policy or another, Jane cannot be provided this service. The difference in our approach is that we have a language designed to express these concerns explicitly at a high level as policies and mechanisms to support the semantics of these policies by appropriately reconfiguring the network.

A number of configuration changes are necessary to provide Jane access when the security policies allow it. In our example, some of the changes involve the Dynamic Host Configuration Protocol (DHCP) [3] server, switches and firewalls in B, and firewalls, file servers and encryption protocols within A, and decryption in Jane's laptop.

Today, most of these configuration changes must be done manually by systems administrators. There is no standard method for performing dynamic reconfiguration of services or network elements in response to changes in the network. Script-based solutions are usually highly dependent on network topology and service/element configuration mechanisms which differ across vendors and even between different versions of the same product. Scripts are frequently highly customized and therefore non-transparent and non-universal.

Moreover, a single change in the network can require changes in multiple scripts, reducing reliability and further worsening the maintenance overhead. Errors in scripts can result in inconsistent network configuration states, and manual recovery made difficult by lack of logging. Moreover, each script must carefully enforce exclusive access to the respective configuration repositories. Another drawback with the lack of centralized meta-information description and repository means that each script must rediscover information, for example network topology, that is not directly instrumented.

Our solution is the first step towards standardizing and automating these configuration changes. First, using NESTOR we build a universal platform to treat network elements in a vendor-independent way. Second, by automating implementation of high-level policy we allow the SA to implement custom policies (which is what she wants) without having to write low-level scripts (which she does not want). The platform is open so that SAs can change existing models or add new ones as necessary to support new policies. Auxiliary services such as logging can also be dealt with at the policy level. The most important aspect of our approach is to ensure that the network remains (at a low level of granularity) in a consistent state; we

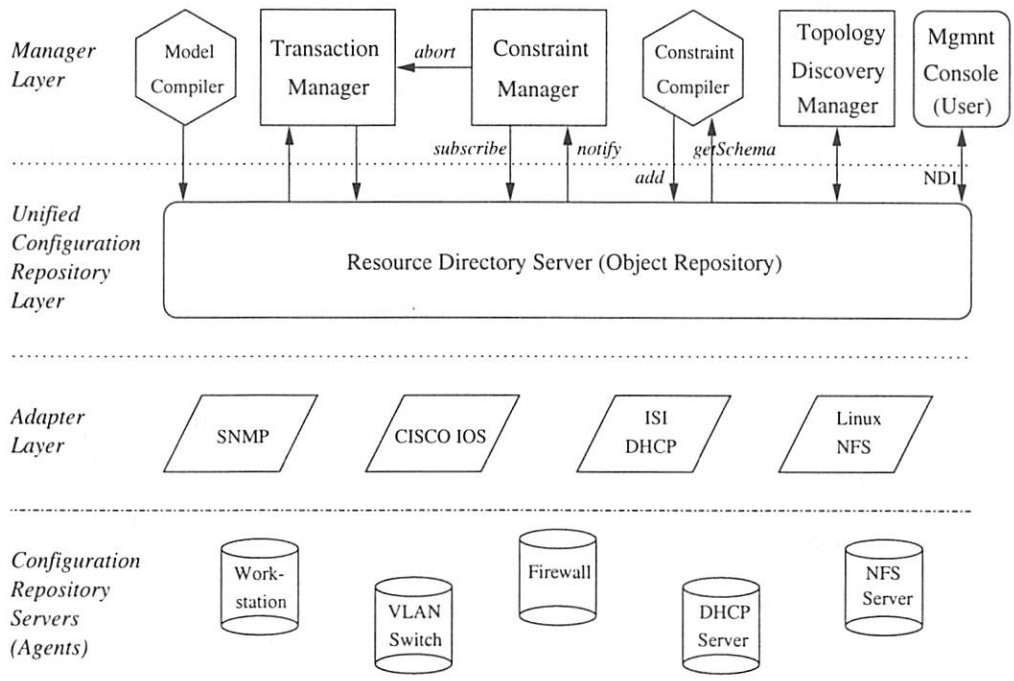


Figure 1: NESTOR Architecture.

achieve this with transaction-like semantics for operations on configuration states.

Our Approach

We use the NESTOR system [16] developed at Columbia University to build an experimental prototype for our experiments. NESTOR is a configuration management system that automates most of the network configuration tasks. Constraints between configuration parameters model interdependence between different subsystems; NESTOR provides support for network-level instrumentation and monitoring of constraints to provide predictable and error-free operations.

In our work we use constraints to also capture security policy. We focus specifically on enabling transparent and secure access in the scenario described in the previous section. This paper describes briefly the architecture of the NESTOR system, how we use NESTOR to build our prototype, how constraints are modeled and managed to provide secure transparent access in a fully automated manner. Future reports will detail progress on high-level security and service policy specifications, compilation to NESTOR constraints, and experiences with larger systems.

NESTOR Architecture

The NESTOR project underway at Columbia University provides a management platform for common network elements. The novelty of the NESTOR manager lies in the fact that it not only automates SA actions but also maintains consistent network state. Consistency is guaranteed by enforcing constraints between various network elements to ensure that change propagation is done correctly. A two-phase commit protocol allows all network changes to be viewed as distributed transactions with rollback features. Our next prototype will handle cases where some network elements do not allow undoing changes. For a general discussion of NESTOR and the details of its architecture please see [16].

Figure 1 shows an overview of the NESTOR system. Systems administrators manage network operations through a unified object-relationship model where managed network elements and services are represented as objects interconnected via a network of relationships. Operationally, the object repository is accessed through the NESTOR Directory Interface (NDI). The interface provides methods for initiating transactions and manipulating repository objects (lookup, create, update, delete). For example, in order to change the name of the host, a systems administrator will look up the matching *host* object in the repository, and modify its *name* attribute. All such updates are performed in the context of a transaction.

Constraints associated with the affected objects are evaluated in order to assure consistency. A constraint is composed of two parts: a declarative predicate and an imperative action. The predicate is evaluated by the constraint manager whenever a change is attempted that may affect its value. When a predicate is violated, the corresponding action associated with the predicate is executed. If the action does not succeed (perhaps because the element being controlled is not responding) the error will propagate so that the system registers the fact that the constraint has been violated and takes appropriate actions to satisfy the high-level policy. Actions have priority levels to manage the relative order in which multiple actions must be executed.

Constraints are first-class objects and are distinguished from other modeled elements. In the host renaming example above, a constraint will state that all host names must be unique. The predicate associated with the constraint will check whether the new name will maintain this uniqueness property. If not, the action will reject the change and may indicate the error on the SA console. If the name change is accepted, the system launches a sub-transaction that may in turn cause other changes as necessary which may launch sub-transactions themselves. In order for the transaction to be committed, the system must reach a consistent state where all constraints are satisfied and all sub-transactions are complete. Otherwise the transaction is aborted and the changes discarded. Unbounded transactions are caught by time-outs and aborted.

Deploying NESTOR

To deploy NESTOR the systems administrator first models the key resources in the network as classes, in the sense of object-orientedness. A model class consists of data members and objects that describe the state and interfaces of the network element. In addition, the model has constraints which define the relationship with other network elements. A detailed example is available in the Network Model section. It is important that the model include all the information necessary to implement the stated policies. Examples include network elements (switches, routers, workstation, servers), network services (HTTP, NFS, NIS, YP, Windows Domains), and optionally policy objects such as a security manager. Common elements models currently available in NESTOR include: Linux (RedHat SysV style configuration) workstation which instruments /etc/* configuration information, SNMP MIB II (to instrument WinNT workstations) and Generic Cisco IOS adapter. Service models in NESTOR include Apache HTTPd, Linux NFSd, ISIC DHCPd, LDAP adapter, Java JNDI and, through the JNDI adapters, YP/NIS access. For the purpose of this work, models of firewalls and other security-related objects were also created. Constraints may be defined at the time of modeling, or added incrementally to the system. Although some constraints may refer to attributes on a single object, most will navigate the object relationships to establish constraints over multiple objects. Examples of modeling and constraint languages are given in the next section.

The next step is to populate the NESTOR repository with model instances. This is done either manually, or in combination with the network topology discovery manager. The topology manager is typically a process executing on a NESTOR server or, possibly, some other network node which scans an IP network for active nodes, and attempts to discover their type (host, router, etc), and services (HTTP, NFS, X11, etc). This discovery process is successful to the extent that the topology manager can only infer information based on SNMP MIB values, port scans, and possibly remote shell commands. For example, a web server may be discovered, but it may not be possible to instrument its configuration unless an appropriate agent (SNMP HTTPd MIB or NESTOR) is installed.

Transactions in NESTOR

Repository transactions are overseen by the transaction manager using a two-phase commit protocol to maintain the transaction properties of atomicity, consistency, isolation, and durability (ACID). There are always at least two participants in every transaction: the initiating entity (e.g., a systems administrator, or the topology discovery manager), and the constraint manager. The constraint manager is responsible for enforcing the constraints stored in the repository. When the transaction initiator requests a commit, the constraint manager verifies that the new state is consistent, that is, all constraints are satisfied. If a violation is detected, the manager will invoke the action associated with the violated constraint. In the case of multiple constraint violations, the order of execution is based on the action priority level, with same level actions executed in arbitrary order. The constraint manager commits the transaction only if all constraints can be satisfied. If a cycle is detected in action execution, the transaction is aborted.

Resource Discovery

The NESTOR Resource Directory Server is responsible for maintaining the network model and constraint object repository. Repository objects implement one or more model interfaces (e.g., *Host*, *HttpServer*). Constraint objects implement the constraints between various objects and thereby encapsulate the mechanism for propagating changes to the underlying resource. For example, an object implementing the *Host* model interface may use SNMP to propagate changes to the name attribute back to the host. Changes are only propagated when a transaction is moved to the commit phase, and are applied in the same order in which the transactions commit (as opposed to the order in which they are initiated).

In order to simplify the task of implementing model interfaces, NESTOR provides adapters for several management protocols and services. For example, the SNMP adapter enables system modelers to map the aforementioned *Host* name attribute to an SNMP MIB object. In addition to SNMP, adapters are provided for the LDAP and NIS protocols, as well as particular implementations of the HTTP, DHCP, and NFS protocols.

Sub-transactions will be ordered so that an aborted transaction does not expose the system. The cost of transactions and the practical implications of system lock-up during configuration changes will be addressed in future reports.

Prototype Implementation

The NESTOR system prototype we used is implemented in the Java language and runtime system. The NESTOR Directory Interface was defined as a Java interface using the Remote Method Invocation (RMI) interface protocol for its underlying communication. The Jini [12] distributed transaction and leasing mechanisms were used in implementing the transaction manager and performing garbage collection at the object repository. Java object serialization was used for persistent repository operations.

The Experimental Testbed

This section describes in more detail the reconfiguration necessary for the scenario of the next section, and presents the specific network employed in our experiment.

Recall that in our scenario Jane, whose home is corporate network A, is now connected to company B's network and wants Web/E-mail/Telnet access to files in her company A. To simplify the exposition, we make a few simple assumptions about the two networks. These assumptions are not necessary in practice; the approach is more general that the example we have chosen to illustrate the capabilities of our management platform. In particular, let us suppose that Company B uses a switched network which supports Virtual LANs (VLAN) and that company A's firewall supports Virtual Private Networks (VPNs) in order to provide remote access to its users over the Internet. Our actual implementation uses Linux for firewalls and hosts and Cisco switches for VLAN support. Further details of the equipment used are provided later in this section.

Requirements for the Scenario

In order to achieve transparent access to the services that Jane wants from her laptop the following configuration changes are necessary:

1. An available Ethernet port will need to be located and Jane's laptop physically connected (in the premises of company B).
2. The laptop will need to be configured for the local network environment, including parameters such as IP address, netmask, DNS servers, default gateways, SOCKS servers, etc. Ideally, this will be achieved automatically using DHCP.
3. In order to maintain Company B's security policy, the laptop will either have to be connected to a special "guest" network and the switch port must be configured for a "guest" virtual

LAN, or all the internal services must be guaranteed to require authentication. The last option is exceedingly difficult to implement in typical networks today and are, in fact, the main motivation for using firewalls to protect corporate networks.

4. Depending on the configuration setting of Company B's firewall, the laptop's address may have to be explicitly allowed to initiate outgoing connections. Company B's security policy may require disabling the laptop's access to any external sites other than Company A since it holds an IP address in Company B's domain. This can be achieved by limiting external connections to the VPN protocol.
5. Once the laptop can reach the Internet, it will need to establish a Virtual Private Network (VPN) connection with Company A's firewall whose policy may be to grant remote hosts limited access to internal resources. Such policies will need to be enforced by all internal services in Company A's network.

Network Topology

The two networks are shown in Figures 2 and 3. For simplicity, Company A's network consists of the internal subnet 172.16.1.0/24 (net-1) and the VPN subnet 172.16.6.0/24 (net-6). VPN clients are allocated addresses from the second network (net-6). The internal Linux NFS server is configured dynamically by the company-A NESTOR server to restrict mobile user access to their home directory. A Linux workstation is used to provide routing, firewalling, and VPN services for the network of company A. The route table is statically set with paths to the internal network, the VPN network, and company B's network. Firewall rules are added to deny all incoming traffic access to the internal network of company A (using Linux Kernel IP Chains). Finally, the CIPE [14] Linux software is configured to enable the remote establishment of a VPN tunnel.

The network of company B is slightly more involved. Instead of an ethernet hub, the internal network is a switched network. Layer-2 switching is provided by a Cisco Catalyst 1900 with support for Virtual LANs (VLAN). The VLAN switch provides for the physical separation between trusted and untrusted IP nodes mandated by the policy of company B. Vlan port assignments will be handled dynamically by the NESTOR server. Company B also has a trusted subnet 10.0.1.0/24 (net-1) and untrusted subnet 10.0.9.0/24 (net-9). Virtual LANs with IDs 1 and 9 physically

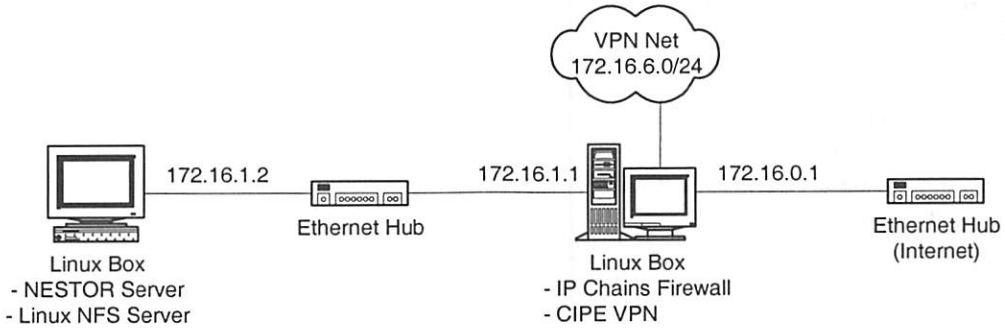


Figure 2: Company A Network.

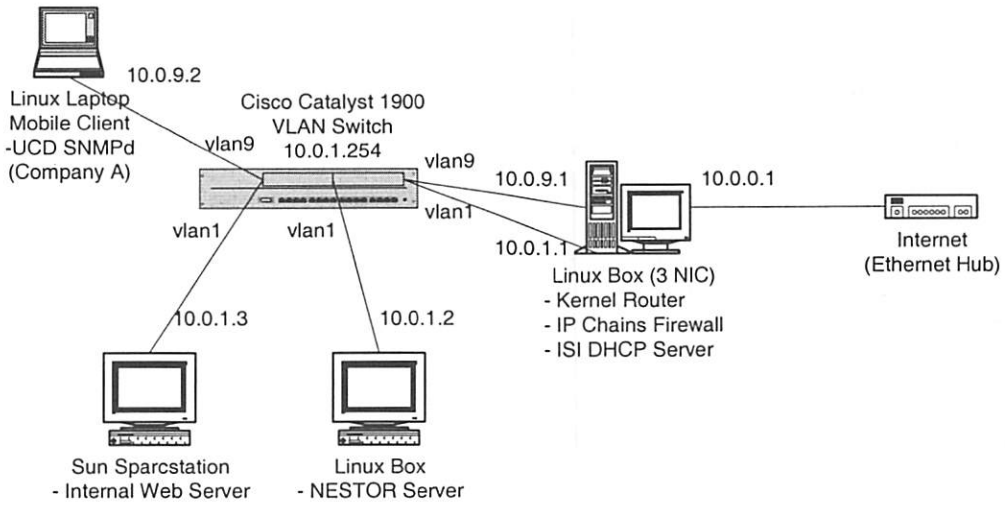


Figure 3: Company B Network.

separate traffic to net-1 and net-9 respectively. A Linux workstation provides routing, firewall and DHCP services to the internal network.

The router has three interfaces, one connected to vlan-1/net-1, another connected to vlan-9/net-9 and the last connected to the external network. Static routes are configured to route traffic from the internal networks (net-1 and net-9) to the external network. Firewall rules prevent any incoming traffic from the external network to the internal network with the exception of established TCP connections to net-9 (guest network). Furthermore, a firewall rule restricts outgoing connections from net-9 exclusively to the VPN port of external network destinations. Obviously, no routing is configured between net-9 and net-1. The DHCP server on the host is configured to listen to the two net-1 and net-9 interfaces for DHCP requests. All unknown hosts are allocated IP addresses from net-9, while a list of trusted hosts (based on their unique client ID which may be their ethernet address) are set to be allocated addresses from net-1. It is assumed that the list of trusted client identifiers is supplied to the NESTOR server. These trusted identifiers will be dynamically configured into the DHCP server by NESTOR.

The networks of the two companies were connected through a common ethernet hub, with static routes configured between the gateways. In this experiment, the NESTOR servers in each company operate independently of each other. The details on how dynamic configuration of the aforementioned networks occurs are discussed in the next section.

Constraints and Automatic Reconfiguration

We now outline how Jane gets transparent access without violating the security policy of either network. The policy of company B not to allow guest machines to gain access to the internal (trusted) network can be translated into the following topology-dependent constraints on device and service configuration. (The next section describes how these constraints can be expressed formally in the NESTOR configuration language).

- Switch ports to which trusted hosts hosts are connected must belong to the internal (net-1) VLAN. Switch ports to which guest (or unknown) hosts are connected must be configured for the guest VLAN (net-9). In cases where internal and guest hosts are connected to the same switch port, the port should be set to the guest VLAN (an alternative would be to disable the port). Violation of this predicate is handled by reconfiguring the VLAN membership of the offending port.
- A firewall rule should prohibit any traffic (including the guest network) from entering the internal network (except for established TCP connections).
- The DHCP server should allocate internal IP addresses only to trusted hosts.

Additionally, company B policy further restricts guest Internet access by limiting guest connections to remote VPN servers. The goal of this policy is to prevent guests from attacking or misusing other networks while in ownership of a company B IP address. This policy is translated into a simple configuration constraint limiting all outgoing traffic from the guest network to well-known ports of VPN services.

The security policy of company A states that remote (VPN) users should receive limited services. In this example, remote users are restricted to accessing only their home directory. We map this policy (there are other ways) to file server configuration by initially denying all directory mounts by VPN hosts. When users sign in with the VPN server to obtain an IP address, permission is added for the particular host to temporarily mount the user's directory. The constraint on file server configuration states that VPN hosts should only be allowed to mount the directory whose user is logged on to the VPN server with that address. This constraint spans the configuration of the VPN server as well as the file server.

In the overall scenario, these constraints (predicates and actions) achieve dynamic reconfiguration in the following manner:

1. User Jane cannot find an available Ethernet port for her laptop, so she "borrows" the network connection of an existing host (thereby obtaining physical access to the internal network).
2. Jane's laptop requests configuration information from the DHCP server,
3. The DHCP server returns a lease on a guest network IP address since the MAC address of Jane's laptop is not included in the DHCP daemon internal network list (at this point the laptop's IP configuration is inconsistent with the link-layer network to which it is connected).
4. By polling the Ethernet switch, the company B NESTOR server discovers the new laptop host. The constraint manager evaluates the constraints which might be violated by the connection of a new host to the switch. The security constraint will be found to be violated, since an unknown (therefore untrusted) host is connected to the internal network. The action component of the constraint is executed, resulting in the reconfiguration of the affected switch port VLAN. Note, that if Jane had connected to an unused switch port, which by default is assigned to the guest network, this constraint would not have been violated. Future prototypes will include mechanisms other than polling.
5. Jane's laptop is now connected to the guest network and will attempt to establish a VPN connection with its home server (at company A) (any other access request, such as web access, is filtered by the firewall of company B).

6. Once the laptop has authenticated with company A's VPN server it is assigned a virtual IP address.
7. The NESTOR server of company A detects this new lease (by polling for the configuration state of the VPN server). The constraint on the file server configuration will be violated since the file server will not be configured to allow home directory mounts from that IP address. This violation will be handled by adding the IP address of the VPN host to the access list of the user's home directory on the file server.
8. After completing her work, Jane disconnects from the VPN server. Again, the company A NESTOR server detects this change, and re-evaluates the affected constraints, resulting in the removal of the file access permission for the previously allocated VPN IP address.
9. After Jane disconnects her laptop from the company B network, and graciously reconnects the host whose connection she had "borrowed", the company B NESTOR server will detect this event and the ensuing constraint violation, leading to the reassignment of the affected switch port back to the internal VLAN.

Network Model and Configuration Constraints

The first step in using NESTOR for our experiment is modeling the network. This section gives some examples of model definitions for network B.

```
interface companyB::EthernetVlanSwitch {
    // relationshipset declares a many-to-many relation between instances
    // of this class, and instances of the EthernetVlanSwitchPort class.
    // The definition includes the role names at both ends of the relation.
  relationshipset consistsOfPorts, EthernetVlanSwitchPort, partOf "a
    many-to-one relation between a switch port to its enclosing switch";
    // An IpInterface is a NESTOR java class maintaining configuration
    // information for IP network interface (IP addresses, netmasks,
    // protocols, etc).
  attribute IpInterface ipInterface;
}
interface companyB::EthernetVlanSwitchPort : CompanyB::Node {
  attribute boolean isEnabled "true if the port is enabled, false otherwise";
  readonly attribute int portNumber "the number of the port in the enclosing
                                     switch";
  attribute int vlanId "the integer Virtual LAN id of this port";
  relationshipset forwardsNodes, EthernetNode, forwardedBy "a many-to-many
    relationship between instances of this class and EthernetNode instances";
  // relationship declares an any-to-one relation between instances of this
  // class, and instances of the EthernetVlanSwitch class. By examining
  // the matching role declaration in the definition of EthernetVlanSwitch,
  // the model compiler determines that this is a one-to-many relation.
  relationship partOf, EthernetVlanSwitch, consistsOfPorts "a one-to-many
      relation between the switch and its ports";
}
interface companyB::SecurityManager {
  boolean isTrusted(int vlanId) "returns true if the VLAN with the given id is
                          considered a trusted network, false otherwise";
  boolean isTrusted(EthernetNode node) "returns true if the EthernetNode is
                considered trusted, false if it is untrusted (or unknown)";
  relationshipset manages, Node, securityManager "a one-to-many relation
          between a security manager and the network nodes it is managing";
}
```

Figure 4: Network rmodel definition example.

Models are expressed in the MODEL language [13] which is an extension of the CORBA [7] IDL with support for relationships, and other features useful for event correlation. Figures 4, 5, and 6 shows a subset of the model definitions for company B.

Consider the EthernetVlanSwitchPort interface definition. This interface models the configuration of a port in an Ethernet switch supporting VLANs. The MODEL definition states that the interface is part of the companyB package and inherits from the Node

```
package companyB;
// [ This code is automatically generated by the model-to-Java compiler.
//   Comments have been manually inserted for clarity. ]
public interface EthernetVlanSwitchPort extends Node {
  public boolean getIsEnabled() throws RemoteException;
  /** returns true if the switch port is enabled, false otherwise. */
  public void setIsEnabled(boolean value) throws RemoteException;
  /** sets the configuration state of the switch to enabled/disabled */
/** model relationships are mapped into Java classes which handle the
    consistent management of relation membership. The declared
    "relationship partOf, EthernetVlanSwitch, consistsOfPorts" is compiled
    into this Java method returning a reference to an object implementing
    the OneToManyRelation interface. The EthernetVlanSwitch interface
    definition will conversely include a getConsistsOfPorts() method
    which will return the same object, but viewed as implementing
    nestor.ocl.ManyToOneRelation. The mapping of model relations to Java
    is an area which will be further explored in the future. */
  public nestor.ocl.OneToManyRelation getPartOf() throws RemoteException;
}
```

Figure 5: Model to Java compiler output example.

```
/** The following is an arbitrarily named Java package where
    implementations of the Java model interface definitions will be
    defined for particular managed elements. */
package companyB.impl;
/** The following class instruments the companyB.EthernetVlanSwitch
    interface for the Cisco Catalyst 1900 switch.  */
public class CiscoCatalyst1900
  implements companyB.EthernetVlanSwitch,
    nestor.adaptor.snmp.SnmpTableListener {
      /** The class constructor is invoked by the topology discovery
        manager, or explicitly by the systems administrator through
        a custom Java program, or in the future through the NESTOR
        GUI.  The implementation further includes code which rediscovers
        the SNMP adaptor after the class has been deserialized (in
        order to support installation in the NESTOR repository). */
  public CiscoCatalyst1900(IpAddress address,
                           SnmpAuthenticationObject snmpAuth,
                           CiscoIosAuthenticationObject iosAuth) {
    nestor.adaptor.snmp.SnmpAdaptor adaptor =
       nestor.repository.Repository.getAdaptor("SNMP");
    adaptor.addSnmpTableListener(target, tableOids, handback, this);
  }
  public boolean getIsEnabled() throws RemoteException { ... }
  public void setIsEnabled(boolean value) throws RemoteException { ... }
  public OneToManyRelation getPartOf() { ... }
}
```

Figure 6: Java implementation.

interface. Three attributes are declared to model: the state of the switch port (enabled/disabled), the port number (a read-only value), and the integer ID of the Virtual Lan to which the port is assigned. The relationship definitions declare a many-to-one relation mapping the port to its enclosing switch, and a one-to-many relation associating the port with the Ethernet (layer 2) nodes which are actively connected to the port.

The EthernetVlanSwitchPort interface represents a device-independent configuration model for an Ethernet switch port supporting Virtual LANs (VLAN). In order to instantiate such an object in the NESTOR repository, an implementation of that interface must be provided with support for the configuration protocols of the actual device being modeled. The next step will therefore be to compile the MODEL interface definitions into a target implementation language. The current NESTOR prototype is built in the Java language and the model compiler converts the extended IDL interface definitions to a set of Java interfaces. As part of the compilation, attribute definitions are converted to a pair of get/set methods (one for read-only attributes) following a simple design pattern. Relationships are compiled into references to collections implementing the OCL [8] (Object Constraint Language) collection semantics. OCL is a language for expressing declarative constraints (side-effect free) and was originally created to state the semantics of the Unified Modeling Language (UML). Unfortunately, due to Java's lack of a parameterized type facility, the translation loses the relationship type information, forcing users to use class casts. An alternative approach of automatically generating new classes for different relation types will be evaluated in the future.

The Ethernet switch supporting VLANs used in this experiment was a CISCO Catalyst 1900 with enterprise edition firmware. The Catalyst supports several SNMP MIBs and may also be configured using a menu system as well as from the command-line. The Bridge SNMP MIB dot1dTpPortTable table was used to instrument the consistsOfPorts attribute of the Catalyst EthernetVlanSwitch implementation. The implementation class registers with the NESTOR SNMP adaptor to receive notification of updates to the table. When a new port is detected, a new instance of the CiscoCatalyst1900Port class is constructed. The port forwardsNodes relationship is instrumented through the Bridge MIB dot1dTpFdbTable. See [2] for details of the components of these tables. The VLAN ID attribute is instrumented using the Cisco IOS adaptor parameterized by the command sequence appropriate for obtaining the VLAN id of this port. The CiscoCatalyst1900Port class, implementing the companyB.EthernetVlanSwitchPort interface, was defined with a single constructor parameterized by the IP address of the managed switch, the switch port number, and an SNMP and IOS authentication object. The authentication objects encapsulate protocol-specific security access information such as passwords and certificates.

Programming Constraints

Based on this model of the network, constraints are defined to maintain the security policies of each domain. To take an example, consider the constraint caused by company B's policy that untrusted hosts should not have access to the internal network. This policy is translated into several constraints on the configuration of network devices. For example, a constraint on the switch states that trusted ports (i.e., those configured for a trusted VLAN) must only be connected to trusted hosts. This constraint, expressed in the OCL language, is shown in Figure 7. In this example, the constraint generates the set of all instances of objects implementing the EthernetVlanSwitchPort interface. The select operator constructs a new set containing only the ports whose isEnabled attribute is true. The forAll operator makes an assertion which must hold for all elements of the selected set of ports. The assertion states that if the port's VLAN is trusted, all the Ethernet nodes which are connected must be also be trusted, otherwise, at least one of them must be untrusted. The checks for size handle the case where no hosts are connected to the port in which case the constraint states that it should be in the untrusted state.

```
EthernetVlanSwitchPort->allInstances
  ->select(port : EthernetVlanSwitchPort | port.isEnabled)
  ->forAll(port : EthernetVlanSwitchPort |
     if (port.securityManager.isTrusted(port.vlanId))
       ( (port.forwardsNodes->size > 0)
        and
         (port.forwardsNodes->forAll(node : EthernetNode |
                        port.securityManager.isTrusted(node))))
     else
       ( (port.forwardsNodes->size = 0)
        or
         (port.forwardsNodes->exists(node : EthernetNode |
                      (!port.securityManager.isTrusted(node))))))
```

Figure 7: A Declarative Constraint: Trusted ports should only forward frames of trusted nodes.

Self-management is achieved by associating a policy script with each constraint. For example, violation of the above constraint, that trusted ports should only forward frames for trusted nodes, may be handled by switching the VLAN id of the port to one which is untrusted. Policy scripts are expressed in an imperative language, which is Java in the current prototype. The policy script is invoked with two parameters, the constraint evaluation stack, and a reference to the transaction object.

The constraint compiler parses the OCL syntax and lists the events which may trigger a violation of the constraint. In the previous example the switch constraint may be violated after the following repository events: create/remove EthernetVlanSwitchPort instance, update in the isEnabled, vlanId, securityManager, and forwardsNodes attributes of an EthernetVlanSwitchPort instance, or changes to the SecurityManager.isTrusted map.

Constraints are first class objects in the repository, which implement the nestor.repository.Constraint interface. When a constraint is written to the repository, the Constraint Manager is notified of the constraint predicate and requests to be listener to the list of relevant events which when triggered may result in a change of state of the constraint.

Constraints are evaluated at the end of a configuration transaction. Transactions may be initiated by a manager, using the repository API, or may be initiated by an adaptor to propagate direct changes to device configuration. For example, the previous constraint example may be violated by a systems administrator if in the course of a transaction a host which was previously untrusted was marked as trusted. The constraint may also be violated when a new (unknown) host is connected to a switch port and this fact is propagated to the repository by the switch adaptor. If an external state change transaction is rejected for any reason resulting in the violation of a constraint, then this fact is propagated back to the affected objects through the relationship network which may find an alternative method of constraint satisfaction (such as an alternative setting of a firewall). If an object cannot find any way of satisfying its constraints, it raises an error message on the SA console.

Populating the Repository

The NESTOR repository may be populated manually or using a graphical user interface that can generate objects given the model and the appropriate parameter values. See Figures 8 and 9. The repository is accessed through a Java Remote Method Invocation API. The API supports methods for adding and

```
public class TrustedPortTrustedHostHandler
  implements nestor.repository.ConstraintHandler {

  // (simple constructor) ...

  /** Handle violation of the constraint that active trusted ports should
      only forward frames of trusted nodes by changing the VLAN id of
      violating ports to the public VLAN id */

public void constraintHandler(Object[] stack, Transaction trans) {
  // Stack: < port, node >

  if (stack.size != 2) throw
    new InternalError("Unexpected stack size=" + stack.size + ": "+ stack);

  EthernetVlanSwitchPort port = (EthernetVlanSwitchPort) stack[1];

  // Obtain the public VLAN id from the security manager of the port.
  port.vlanId = port.securityManager.getPublicVlan();
}
```

Figure 8: Trusted port/host handler.

```
package nestor.repository;

public interface Constraint extends java.io.Serializable {
  public void checkConstraint(Repository repos, Transaction trans)
              throws ConstraintException;
  public void checkConstraint(RepositoryEvent[] events, Repository repos,
                      Transaction trans) throws ConstraintException;
  public RepositoryEvent[] getConstraintEvents();
}
```

Figure 9: NESTOR Repository.

removing objects, locating objects based on the class and attributes, and initiating transactions. To add an object to the repository a systems administrator initiates a transaction and then adds the object within the transaction. The object must implement one or more model interfaces and support the serializable interface (i.e., may be stored as a byte string for transport over the network). Storage in the repository is provided on a lease basis which must be renewed by some entity such as a lease renewal manager, or the object itself. If a lease expires an object may be killed or archived if possible. If there are constraints that may be affected, an error message is raised on the console. This obviates the need for vigorous garbage collection. The current NESTOR prototype utilizes the Jini [12] distributed leasing, event, and transaction APIs.

The repository can also be populated with the help of a utility for topology discovery. The utility executes on a host, and periodically pings each network address to establish a map of active nodes[1]. Currently, our topology manager accepts classless IP network and netmask combinations. Once a node is detected as being active, the utility attempts to extract information using the SNMP protocol, and tests for service availability by attempting to connecting to different services (such as Telnet, HTTP, NFS, FTP, etc). In its current incarnation, the topology manager is mostly focused on discovering workstations (such as Linux and Windows NT boxes) and supplying information about their interface configurations, route tables, and active services.

Returning to our scenario, the administrator constructs a new instance of the CiscoCatalyst1900Switch object using the IP address assigned to the management interface of the VLAN switch and the appropriate authentication information for administering the switch which are the SNMP community and IOS passwords. NESTOR repository objects implement an initialization and control interface (analogous to Java applets) so that their execution can be controlled by the NESTOR server. An object may query the repository for services such as adapters using an instance of the RepositoryContext interface. For example, the switch object will use an SNMP and IOS adaptor instances. New adaptors may also be used provided they implement the NestorAdaptor interface). After obtaining the necessary adaptor references, the object will subscribe for notification of changes in the relevant SNMP objects, and IOS results.

When the administrator commits the transaction to create the new switch object, the transaction manager will verify that the addition did not violate any constraints. The constraint, shown in Figure 7, may be violated when new instances of objects implementing the EthernetVlanSwitchPort interfaces are created. Assume the initial switch state does not violate the aforementioned constraint. When user Jane connects her laptop computer to network B, and in particular to a switch port, the switch SNMP bridge MIB table dot1dTpFdbTable will add the laptop's MAC address. At the time of the next poll by the NESTOR SNMP adapter, the change will be detected resulting in notification of the subscribing

CiscoCatalyst1900Switch object. The switch object will look up for an instance of EthernetNode with the same MAC address, creating a new instance if one is not found. The EthernetNode is then added to the switch's forwardsNodes relation. At the point where all propagated changes have been reflected in the model, the switch object will commit the changes. At this point the constraint manager will again verify the set of constraints which may have been affected by the transaction. In this example, since Jane connected her laptop to a switch port previously assigned to the internal network, the constraint on switch port VLAN state will be violated, and the policy script will be executed as outlined earlier in the paper.

Future Extensions and Applications

As mentioned in the introduction, the work described in this paper is the first step towards our long-term goal of building a management platform for security and service availability. Several interesting theoretical and implementational issues have been identified and examined in this project. Foremost amongst them is the need for formal tools to express network security and service policy. In the current work, we have assumed that the two networks in question have security policies that have to be obeyed when reconfiguring the network. Upcoming reports will deal with the questions of how general network security policy can be stated formally and implemented automatically by integrating it with NESTOR with appropriate translation mechanisms. This requires settling questions of an appropriate language for security policy and checking consitency of policies.

Another dimension of our approach that we did not address in this paper is scalability. For the network management platform to be viable in wide-area networks, management discipline must be lightweight and modular. Furthermore the response time of the management software to proposed changes in the network must be fast enough to keep pace with the rate of change. Our current architecture uses centralized NESTOR servers but in a large network this is likely to be infeasible. To ensure fast response times, some of the management functionality must be localized. Local decision-making capabilities have to be balanced against the more important goal of automatic network consistency. Since verifying (and consequently, propagating) each and every change in a dynamic network with a central global policy server is not likely to be successful, our approach is to distribute the task of maintaining global network

[1]The security warnings that these may generate will have to be handled.

properties such as consistency and security policy using constraints between network elements. We plan to investigate a distributed architecture for our management platform wherein every network element such as a switch or laptop has some NESTOR functionality within it so that network elements can directly communicate with each other using a NESTOR-like interface which implements constraints between these elements. For example, a switch may be enabled to reconfigure within a predefined set of allowed configuration (as defined by the global security policy) in response to local changes. Our future work on scalability will address the problem of partitioning policy and constraints in a network between its various elements.

In the short term, the current design of NESTOR will be extended with a focus on the design of the security, distribution, replication and caching protocols, as well as optimization issues in the current prototype implementation. NESTOR will be applied to verifying the configuration of the Columbia University department of Computer Science. Deployment on a large, live network such as Columbia CS will help identify areas for performance optimization. The NESTOR source code and sample models will also be made freely available for downloading[2].

Related Work

The most closely related management architecture to NESTOR is the ICON system [5] which used active database style Event-Condition-Action (ECA) rules to state restrictions on objects instrumented by SNMP MIB values. The NESTOR system also incorporates services such as multi-protocol access to heterogeneous resource information, configuration transactions, declarative constraint, and constraint propagation through policy scripts. The Dolphin project [9] developed a declarative language for modeling network configuration and operation for fault analysis where the emphasis was on deducing the cause of failures that have occurred by tracing the propagation of operational rules in the model. Constraint-based management has been pursued previously in [10] and [11] where constraints are employed for the diagnosis of network faults. In the area of configuration management automation, the GeNUAdmin system [6] is an off-line tool for extracting network configuration information into a centralized database, performing updates on that database which are checked for consistency, and pushing the changes back to their respective configuration files. Simple consistency checks are performed to assure that added values are valid and that key values are unique. The RPI service dependency tool [4] detects service dependencies and generates up to date server listings. The goal of the system is to prevent unforeseen service interruptions caused by hidden service dependencies. Ganymede [1] is an extensible and customizable directory management framework applied to the central management of user and host data, which is distributed in different databases. Ganymede supports transactions on the central repository objects, but does not provide a constraint mechanism beyond a few built in security, and deletion propagation checks. NESTOR can support these functionalities given an appropriate set of constraints on the unified configuration model.

[2]See http://www.cs.columbia.edu/dcc/nestor.

Conclusions

Providing guest users secure network services requires automatic, dynamic, and safe configuration of multiple devices and services. Even though the operational and security constraints span multiple devices, services, and profiles, NESTOR provides a unified model of network configuration which significantly simplifies specification and management of network constraints. By separating the model definitions from the instrumentation layer implemented by NESTOR, we can implement high-level security policies automatically and effectively by compiling them into NESTOR constraints on network elements and services. Constraint resolution is achieved through the automatic execution of policy scripts whose actions are subject to the constraints defined. This, combined with the fact that all configuration changes are logged, promises to make use of automated reconfiguration a practical reality.

Acknowledgments

Part of this work was done when Alexander Konstantinou was a summer intern at Telcordia. The NESTOR project at Columbia University is sponsored by DARPA Contract DABT63-96-C-0088. The project on self-management of security properties at Telcordia Technologies is sponsored by DARPA Contract F30602-99-C-0182. We thank Eric Anderson for his insightful comments on an earlier version.

Author Information

Sandeep Bhatt <bhatt@research.telcordia.com> is Senior Research Scientist at Telcordia Technologies. His current research involves self-configuring network management systems. His previous work has included network monitoring and fault-management systems, high-performance distributed simulations of communication networks and physical N-body systems, theory of graph embeddings, with applications to VLSI circuit layout and parallel computing. He received his S.B., S.M., and Ph.D. degrees from the Massachusetts Institute of Technology in 1978, 1980, and 1984 respectively. He was previously Associate Professor of Computer Science at Yale University, and in 1990 Visiting Associate Professor of Computer Science at California Institute of Technology.

Alexander Konstantinou <akonstan@cs.columbia.edu> is a doctoral candidate in the Computer Science

Department at Columbia University in the city of New York. He was awarded the M.S. degree in computer science from Renselaer Polytechnic Institute (RPI) in Troy, NY (1996), and a B.A. in history and computer science from Macalester College in St. Paul, MN (1994). His research interests include computer networks, network management, and distributed systems. Currently, he is leading the NESTOR project at Columbia University under the supervision of Professor Yechiam Yemini. He has previously been employed as systems administrator managing the Unix laboratory in the Macalester College Mathematics and Computer Science department.

Sivaramakrishnan Rajagopalan <sraj@research.telcordia.com> is a Research Scientist at Telcordia. His Ph.D. thesis at Boston University devised secure ways of accelerating block ciphers. His research interests include cryptography, security of web applications, and traffic and fraud analysis. He holds patents on algorithms for provably secure video-rate encryption, and efficient, data-compression. These are used by phone companies to compress logging data on telephony networks. Together with colleagues in 1995 he found security flaws in firewalls which could be exploited by Java programs. This attack was widely reported and has influenced security administrators strongly on the dangers that downloaded executable content pose for protected networks. He has published in the fields of Cryptography, Data Compression, Internet Security, and Complexity Theory.

Yechiam Yemini <yemini@cs.columbia.edu> is Professor of Computer Science at Columbia University. His research interests include computer networks, network management, high-speed networks, and protocols. He has authored over 150 publications and lectured extensively in these areas. Research at his Distributed Computing and Communications (DCC) laboratory resulted in widely exported network software technologies applied by hundreds of sites and commercialized by several companies; see http://www.cs.columbia.edu/dcc. He was a co-founder of Comverse Technology Inc., the lead vendor of voice mail systems for telecom networks, see http://www.comverse.com/, and of System Management Arts, Inc., a DCC lab spin off producing software products to automate event correlation and fault diagnosis in networked systems; see http://www.smarts.com.

References

[1] J. Abbey and M. Mulvaney, "Ganymede: An extensible and customizable directory management framework," in *12th USENIX System Administration Conference (LISA '98)*, 1998, http://www.arlut.utexas.edu/gash2/.

[2] E. Decker, P. Langille, A. Rijsinghani, and K. McCloghrie, "Definitions of managed objects for bridges," Tech. Rep. RFC 1493, IETF, July 1993.

[3] R. Droms, "Dynamic host configuration protocol," Tech. Rep. RFC 2131, IETF, Mar. 1997.

[4] J. Finke, "Automation of site configuration management," in *11th USENIX System Administration Conference (LISA '97)*, 1997.

[5] S. K. Goli, J. Haritsa, and N. Roussopoulos, "ICON: A system for implementing constraints in object-based networks," in *IFIP/IEEE Integrated Network Management, IV*, 1995.

[6] M. Harlander, "Central system administration in a heterogeneous unix environment," in *8th USENIX System Administration Conference (LISA VIII)*, 1994.

[7] Object Management Group, "The Common Object Request Broker: Architecture and specification," Tech. Rep. Revision 2.3, OMG, June 1999.

[8] Object Management Group (OMG), "Object constraint language specification," tech. rep., OMG, 1 1997.

[9] A. Pell, K. Eshgi, J. J. Moreau, and S. Towers, "Managing in a distributed world," in *IFIP/IEEE Integrated Network Management, IV*, 1995.

[10] M. Sabin, R., D. Russel, and E. C. Freuder, "Generating diagnostic tools for network fault management," in *Fifth IFIP/IEEE International Symposium on Integrated Network Management (IM'97)*, 1997.

[11] M. Sabin, A. Bakman, E. C. Freuder, and R. D. Russel, "Constraint-based approach to fault management for groupware services," in *International Symposium on Integrated Network Management*, 1999.

[12] Sun Microsystems, "Jini architecture specification," Tech. Rep. version 1.0, Sun Microsystems, Jan. 1999.

[13] System Management Arts (SMARTS), "Event modeling with the MODELlanguage: A tutorial introduction," tech. rep., SMARTS, 1996. http://www.smarts.com/.

[14] O. Titz, "Cipe – crypto ip encapsulation," tech. rep., inka, 1999. http://sites.inka.de/%7eW1011/devel/cipe.html.

[15] Y. Yemini, A. V. Konstantinou, and D. Florissi, "NESTOR: An architecture for self-management and organization." To appear in IEEEJSACspecial issue on network management.

[16] Y. Yemini, A. V. Konstantinou, and D. Florissi, "Nestor: An architecture for self-management and organization," tech. rep., Dept. of Computer Science, Columbia University, Sept. 1999. http://www.cs.columbia.edu/dcc/nestor/.

It's Elementary, Dear Watson: Applying Logic Programming To Convergent System Management Processes

Dr. Alva L. Couch and Michael Gilfix – Tufts University

ABSTRACT

In an ideal world, the system administrator would simply specify a complete model of system requirements and the system would automatically fulfill them. If requirements changed, or if the system deviated from requirements, the system would change itself to converge with requirements. Current specialized tools for convergent system administration already provide some ability to do this, but are limited by specification languages that cannot adequately represent all possible sets of requirements. We take the opposite approach of starting with a general-purpose logic programming language intended for specifying requirements and analyzing system state, and adapting that language for system administration. Using Prolog with appropriate extensions, one can specify complex system requirements and convergent processes involving multiple information domains, including information about files, filesystems, users, and processes, as well as information from databases. By hiding unimportant details, Prolog allows a simple relationship between requirements and the scripts that implement them. We illustrate these observations by use of a simple proof-of-concept prototype.

Introduction

Lately, the task of system configuration has been greatly eased by tools that automatically enforce compliance with a model of proper system operation and health via 'convergent processes' that detect and correct deviations from the model. System management then becomes a matter of crafting the model of appropriate behavior or configuration. This model contains 'rules' that specify proper behavior and configuration together with 'actions' that specify what to do to correct any discovered lack of compliance with a given rule.

Unfortunately, crafting such a model is difficult due to the number of different kinds of rules and actions involved in creating a complete model. These range from high-level operating policies to specification of dynamic operating behavior. First we must specify *operating policy*, a high-level description of how the system should behave and what services should be offered. We must then translate this high-level behavioral description to a description of the contents and disposition of system files that will insure this behavior. We can call this description the *static configuration* of the system, because the requirements it describes should not change over time, and typically we can insure these requirements are met with a single script or scripts, executed once. The system then begins operation and interprets these files in order to operate, so that other content does change over time. We can call the things that change over time part of the *dynamic configuration* of the system. To conform this to our requirements, we can craft *convergent processes* that observe the dynamic configuration of the system and modify system performance to match our models.

Static Configuration

There are now an endless variety of tools available for incrementally assuring desired static configuration of a system or network, beginning with the legacy of make [22] and rdist [8], both of which control file state based upon incremental generation and copying rules. The ideas in these tools are now pervasive and have made their way into almost all tools for configuration management. Package managers such as the RedHat Package Manager (RPM) [2] and Depot [7, 21, 28] only install requested software packages if those packages are not already present. Our own tool Slink [9, 10] and its relatives, including GNU Stow [14], incrementally modify a symbolic link tree to conform to a desired structure, while our own tool Distr [11] allows 'push' and 'pull' convergent file distribution, utilizing filters to translate file formats for differing platforms.

Cfengine [3, 4, 5] makes it possible to define and converge to very complex and expressive models of system state. Cfengine provides a powerful configuration language with built-in operations that act on files, links, directories, mounts, and even processes. Extensive built-in stream editing commands allow us to incrementally edit system files to conform with requirements, freeing us from having to store file prototypes on a master server.

All these static configuration tools share the same strengths and limits. Configuration files are relatively simple and easy to construct. The process by which one assures conformance with a configuration

file is obvious and automatic. However, with the exception of a small number of database-driven prototypes, these tools specify system configuration at a fairly low level of abstraction, telling what to do to specific file contents.

One would like, instead, to simply list desired services and have the tool determine what to place in each file to implement each desired service. An ideal tool would query a distributed database or directory service, such as the Lightweight Directory Access Protocol (LDAP), for a high-level description of the services required on the system in question. Based upon the list of services to be offered, the tool would then proceed to modify all files requiring changes in order to provide that service. This kind of configuration power would require that the configuration tool know the mapping between services and file contents for each target operating system. This, in turn, requires maintenance of rather simple, detailed databases of system information that have little or nothing to do with operating policy: where files are, how configuration files are structured, etc.

Dynamic configuration

Historically, dynamic configuration has been preserved and enforced by a completely different set of tools than those used for managing static configuration. While static configuration tools rely heavily on databases, lists, and other declarative mechanisms for specifying configuration, dynamic configuration tools have relied upon user-crafted scripts. To use a tool, the administrator specifies behavioral patterns to detect and scripts to execute when each pattern is detected.

Current dynamic configuration tools allow monitoring of dynamic state, including processes, logfiles, and filesystems. Early tools were system log monitors, such as Swatch [15] and the more recent LogSurfer [18], which can page operators or run other scripts when potentially harmful events are posted in the system logs. These simple monitors have evolved into powerful tools that can monitor global system state, including TripWire [16, 17], which checks whole filesystems for compliance with a previous recorded state, and SyncTree, which can restore previous states of a system even if they are changed maliciously by a hacker [19].

Many system administrators resign themselves to writing custom scripts to monitor and correct problems in UNIX networks. These scripts interact with the same UNIX commands, and perform the same tasks, but must be customized for each site and platform, leading to massive duplication of effort. PIKT [24] (pronounced 'picket') greatly reduces the effort in writing scripts for multiple platforms and tasks, by providing a class mechanism for determining applicability of script parts and a powerful set of built-in parsing primitives (reminiscent of command parsing available in Tcl/TK [23]) for accessing the text output of UNIX status commands such as ls and netstat. Similar scripts for different platforms can be organized into a single script with class qualifiers, where appropriate lines in the script will be utilized for each target platform.

PIKT works well but, like the custom scripts it allows one to catalog, there is an uncomfortable distance between the scripts that implement policy and the policies they implement. A policy that is relatively simple to describe, such as "delete all core files more than three days old" might be written as the find command:

```
find /home -name core -mtime 3 \
        -exec rm -rf {} -print;
```

or something even more esoteric. It can be quite difficult to work backward from an arbitrary script to its meaning.

Ideally, we should be able to document operating policy and automatically translate the documentation into scripts that implement the policy. Ironically, the typical administrator in a hurry will document only the scripts. To determine what operating policies are, one must read and interpret what the scripts mean. Most administrators are not paid to write scripts, but to insure quality of service, so that script writing is done in great haste and with no attention to readability or potential software life-cycle. So documenting the actual operating policy for a network requires reverse-engineering the operating policy from the scripts on an ongoing basis.

Again, we need some way to automatically translate between a high level description of operating policy and the script that implements it, so that we no longer have to read and understand a script to understand the policy it implements. There must be a way to craft scripts that is in some way 'closer' to the natural way we would describe policy: a language closer to specifying what we want rather than how to accomplish it.

Databases

Many administrators have come to rely on databases [12], both normal and directory-based (such as LDAP or NIS+), to describe static network state. A database is a structured data storage and retrieval method, consisting of tables of information, where each table is organized into rows and columns. Database information is usually manipulated and accessed by use of Sequential Query Language (SQL), which specifies how to access individual rows and columns, and how to create new tables whose rows and columns can then be accessed.

Databases have several advantages over plain files. They can be accessed from anywhere in a network using standardized network access methods. Isolated parts of a database table can be incrementally modified with no chance of corrupting other parts of the table. When information is volatile, but must be modified and accessed in small chunks, databases provide more reliability than unstructured files.

Databases become very useful in maintaining information about the *external* world outside the systems being maintained, such as information on each user's true identity and function. A typical application would be to record information about each user in a database, and then use that information to compute appropriate filesystem and mail quotas for the user.

Unfortunately, normal database access methods such as SQL do *not* allow one to specify how to *act* based upon database contents. One must call SQL from another language empowered to take action. So to use databases, we are forced to learn both SQL *and* a scripting language (such as Perl [26]) for crafting actions based upon SQL queries. How, then, can we utilize databases without having to simultaneously write in two scripting languages: one for database access and another to react to content?

Toward a 'Glue Language'

We seek to fulfill Burgess' dream of 'Computer Immunology' [5], in which a description of computer 'health' empowers computers to 'immunize' themselves against poor function, thus becoming self-repairing and correcting. We began the work of this paper by searching for a common language that one could use for specifying both static and dynamic configuration. If we could find such a 'glue language,' we would be one step closer to being able to write scripts that describe operating policies with true platform independence. The language had to be able to provide at least a superset of the combined capabilities of Cfengine and PIKT. As well, we desired a language that:

- allows both static and dynamic requirements, limits, and convergent processes to be specified with the same syntax.
- is extensible to provide interfaces to all conceivable kinds of data and actions.
- allows specification of high-level rules that codify all steps in providing one user service, so that users can simply ask for the service rather than describing its low-level modifications.
- interoperates well with structured forms of information storage such as databases and directory services.

We came to a surprising conclusion, even for us, that the closest existing language fitting that description is Prolog!

Prolog As a Database Query Language

Prolog [6] is a much misunderstood language with an somewhat undeserved reputation for inefficiency and difficulty of programming. In reality, Prolog is one of the most efficient mechanisms for making *queries into databases* and writing action scripts based upon database queries. As well, Prolog has unique *implicit* properties that make programs shorter and easier to read, by omitting details that the language can handle by itself. For example, both conditional statements and loops are implicit in Prolog, and their use is determined by context.

We explored the powers of this language by constructing a prototype interface between Prolog and the operating system on a single host, with the intent of creating Prolog utilities that duplicate the functionality of Cfengine and PIKT. Then, we experimented with the prototype to determine its strengths and weaknesses.

Prolog syntax

Programming in Prolog is very different from programming in a normal scripting language. Rather than saying what should happen, one declares what *should be true*. The Prolog interpreter translates those declarations into actions to perform. This is called *declarative programming*.

A Prolog 'program' consists of *facts* and *rules*. A *fact* can be thought of as a line entry in a table in a database. The fact:

```
login(couch).
```

says that there is a user whose login name is couch. It has a *functor name* of login and a single *argument* couch.

Facts can be pre-recorded in Prolog's databases, or can be computed by external functions written in C or other languages. For example, in our prototype, we compute facts of the form

```
passwd(couch,
  '3hit2839482912',
  1000,
  40,
  'Alva L. Couch',
  '/home/couch',
  '/usr/bin/tcsh').
```

with an external function (written in C) that scans the password table (as an NIS+ map) and reports its contents. This function implements the *functor* passwd of *arity seven* (seven arguments). This functor is named passwd/7 to distinguish it from other functors with the same name and differing numbers of arguments, which need not be related to it.

A Prolog *rule* tells how to make more complex facts from simpler ones. For example, the rule:

```
pig(Login):-
  passwd(Login,_,_,_,_,Home,_),
  du(Home,Usage),
  Usage>20000.
```

says that "Login is a pig if Login is a login name with home directory Home, Usage is the disk usage for that directory, and the disk usage is greater than 20 megabytes (approximately)."

A rule has a left hand and right hand side separated by :-. The left-hand side specifies the goal of the rule, which in this case is to find a value for the variable Login. The right-hand side consists of subgoals

needed to accomplish a goal. The symbol :- is read 'if', and commas between terms on the right hand side represent 'and'. Login, Home, and Usage are *variables* because they begin with capital letters. The special symbol "_" (the *anonymous variable*) is a place holder that indicates that a value in a query should be ignored. In this rule, for example, we can ignore the user's password, uid, gid, name, and shell.

Queries

In order to get Prolog to actually do anything, one has to execute a *query*. This is a request to compute values of variables based upon known facts and rules. For example, to request a list of pigs, from the rule above, we could type this in the Prolog interpreter:

```
?- pig(X).
```

Prolog might respond:

```
X=couch ;
X=bgates ;
No.
```

The symbol ?- can be read as 'prove'. This query instructs Prolog to "find all X's such that pig(X) is true." Prolog responds with the first of these, couch. After each response, we type a ";" to tell Prolog to find the next value for X. The final No. indicates that there are no more matches.

Whenever Prolog needs to determine who is and who is not a pig, it uses the rule above to *compute* who all the pigs are. Prolog begins with no idea of who a pig is, and evaluates the subgoals in the rule on the right hand side of the 'if' from left to right. The first subgoal, passwd(Login,_,_,_,_,Home,_) sets Login to each login name in turn, and sets Home to the corresponding home directory of Login. Then the second term, du(Home,Usage) *computes* the disk usage for that directory (by scanning the home directory) and sets Usage to that value. The last subgoal, Usage>20000, checks the value Usage. If it is greater than 20000, the goal pig(Login) *succeeds*. This has the result of returning whatever Login value we found as the result of the query. In this case, we wanted X's, so each match is assigned to X and printed for us.

Backtracking

Queries are repeatedly satisfied through *backtracking*. To backtrack, Prolog backs up from right to left in the list of subgoals it is attempting to complete, and tries new values for variables. For example, we implemented passwd/7 so that when backtracking, passwd(Login,_,_,_,_,Home,_) will set Login and Home to the information for each user in the system, one per try. Through backtracking, the rule above can potentially check 2000 users for pigdom. After finding a new value for Login and Home, Prolog then continues trying to execute goals from left to right. Whenever it satisfies all subgoals of a goal, and gets to the end of the rule, Prolog returns a match.

In this way, Prolog enumerates all possible matches for each rule, as demonstrated above. Every Prolog goal potentially tries all reasonable values for each variable, so that one never has to write a 'for' loop in Prolog. This also means that the easiest program to write in Prolog is an infinite loop!

Backtracking is as tricky and dangerous as it is powerful. Suppose that instead of the preceding rule for pig/1, we wrote:

```
pig(Login):-
  du(Home,Usage),
  passwd(Login,_,_,_,_,Home,_),
  Usage>20000.
```

This never succeeds in finding any pigs, because Home and Usage are unbound when du(Home,Usage) gets called. The passwd/7 call must go first.

Performance is seriously affected by the way in which we write a Prolog rule. The original pig/1 rule finds all the pigs in a system roughly as quickly as a Perl script written to do the same thing. But suppose we instead write:

```
pig(Login):-
  passwd(Login,_,_,_,_,_,_),
  passwd(Login,_,_,_,_,Home,_),
  du(Home,Usage),
  Usage>20000.
```

This *means* the same exact thing as the original rule; the first subgoal requires that Login be a login name, while the second requires that Home be the corresponding home. Depending upon the cleverness with which we implement passwd/7, however, executing this query can take between twice and thousands of times as much time to execute, compared to the original rule.

Our first version of passwd/7 read and cached the whole NIS+ password table before returning each entry. This meant that the rule above did that *twice*. First, it backtracked through all values for Login, and then checked them against all values for Login in order to find a matching one and determine its Home. Since we have about 1000 users, the rule thus checked 1000 entries 1000 times in backtracking to satisfy both subgoals. This took forever.

We then re-implemented passwd/7 so that if Login is initially bound to a value, passwd/7 uses getpwnam rather than getpwent to match information instantly. The second subgoal passwd(Login,_,_,_,_,Home,_) thus does *not* try all combinations, but instead instantly returns the appropriate home directory. This change made the above example execute several thousand times faster, but it still takes about twice the time of the original, more efficient rule with one subgoal for passwd/7.

Unification

In converging toward system health, we wish to force our idea of what should be true to actually be true. In Prolog nomenclature, we wish to *unify* our

idea of what reality should be with the state of a particular machine.

In a Prolog rule, variables begin their lives having no value whatsoever, and are called *unbound.* Variables become bound, or set, by being *unified* with constants or other variables. The way this works depends upon how built-in and external functions are designed, and functions can behave differently depending upon whether variables given to the function are bound or unbound. In evaluating the goal:

```
passwd(Login,_,_,_,_,Home,_)
```

much depends upon the *prior state* of the variables Login and Home before the goal executes.

1. if Login and Home are unbound, then the query tries to bind Login and Home to each valid pair in the password table (in our case, via NIS+ naming service).
2. If Login is bound but Home is unbound, then Home is unified with the corresponding home directory, if any.
3. If Home is bound but Login is unbound, then Login is unified with each login name with that home directory in turn (and there may be more than one)!
4. If Login and Home are both bound, then the query 'succeeds' if they are a valid pair, and 'fails' if they are not paired correctly.

The result is that after any 'success', Login and Home form a valid pair, regardless of how that pair arose. A general-purpose goal like this can be used in many contexts, with variables known and unknown, and will adapt to the context and respond appropriately. Given a small amount of information, of any kind, this rule determines the rest if possible. Prolog goals (the left hand sides of rules) are called *functors* (rather than functions) precisely because they are capable of setting variables in a very flexible way, and using the same variables as both input and output.

Goals That Modify Configuration

In using SQL to manipulate tables of configuration information, one must use a different language for actions than for queries. Not so in Prolog. There is no reason that a goal cannot modify the external world in order to satisfy a query. Consider the simple example of a goal that returns the owner and group of a file specified as a pathname:

```
path_owner(Path,Owner,Group).
```

This could be implemented in Prolog in several ways. For simplicity, and to avoid implementing too many external functions, we chose to implement the goal so that unbound attributes for a path are read from the filesystem, while the filesystem is *modified* to match bound attributes if possible. The query

```
?- path_owner('/etc/motd',
               Owner,Group).
```

unifies Owner and Group with the true owner and group of the file /etc/motd, and reports them, while the query

```
?- path_owner('/etc/motd',0,0).
```

attempts to *change* the file's owner and group to 0 (root) if they are not already both 0! The same function serves a dual purpose: it can *query* system state or *modify* that state to *unify it with specifications.* Thus unification need not only concern Prolog variables, but can be extended to modify the environment in which Prolog executes!

The ambiguity between goals and actions in Prolog can be exploited to construct many rules that implement both at once. If we write a goal copy(Source,Target) that insures that the file Source is identical with node Target, then goal succeeds if it *can make the files identical* and fails if it cannot. Then we can write:

```
?- copy('/Master/etc/motd',
        '/etc/motd').
```

to check and perhaps make a copy of a master file. Goals like this, which blur the distinction between doing something and checking it, are ideal for use in creating a configuration engine.

Such lingual power comes with a price. Structured programmers should cringe, because we have all been taught that programs should not contain 'side-effects' that change things other than program variables. In the above, we are executing 'ambiguous' goals that check and assure system states *solely for their side effects!* This can make Prolog programs difficult to interpret and maintain.

Careless implementation and use of convergent Prolog goals can lead to disaster. Suppose that we implemented the passwd/7 functor so that it was able to *set* any field to a desired value, as a side effect, as well as iterating over all password records. Then the query:

```
?- passwd(_,_,30,_,_,_,_).
```

would set everyone's user ID to 30! For our peace of mind, we have refrained from writing functors that are both iterators and convergent modifiers. Each of our custom functors either enumerates options or tries to assure individual conditions, not both!

Implicit Iteration

Suppose we want to tell all pigs what we think of them. In Prolog, iterating over all matches for a variable is *implicit.* One never has to write a for or foreach loop; any query will search through all possible options. For example, if we type:

```
?- pig(Login),
   email(Login,
         'you are a pig!',
         'oink!'),
   fail.
```

Prolog will mail the message 'you are a pig!' (with subject 'oink!') to all possible pigs, as determined by the rule above! First the goal pig(Login) tries to find a pig.

When one is found, the goal email mails a message to that pig. The magical thing here is the built-in Prolog goal fail. This goal *forces Prolog to backtrack* through all possible solutions to the preceeding goals. Thus *all pigs* get the message!

Implicit execution is both a curse and a blessing. The good news is that one never needs to write a 'for' loop again. But interpreting what Prolog code actually does can be difficult. A safe way to interpret a Prolog program is to mentally put the phrase "for all values of variables so that" in front of every goal.

Controlling Implicit Loops

Sometimes we may wish to *prohibit* this behavior, e.g., make an example of *one* pig. The special goal ! (the *cut*) tells Prolog *not* to try to backtrack to the goals on its left-hand side. We could type:

```
?- pig(Login),!,
   email(Login,
         'you are a pig!',
         'oink!'),
   fail.
```

This would find one pig, but the fail would not cause backtracking to find others. The cut's general meaning is actually a bit more subtle: when backtracking over a cut, the *parent goal* fails. The cut operates on the *problem* Prolog is trying to solve (in this case, the whole query), not the sequence of goals being utilized to solve the problem.

Prolog and Cfengine

So far we have shown only how to construct specific queries that have particular effects, by manually interacting with the Prolog interpreter. How, then, does one automate the process of configuring a complete system? As with Cfengine, we must craft a set of rules that describe when the system is healthy. We can learn much from how Cfengine rules accomplish the same task.

Cfengine is 'almost as powerful' as a real Prolog interpreter, and only falls short in its handling of variables and extensibility. To control whether and how to do something with a particular machine, Cfengine uses 'classes'. A 'class' is a variable that is either true or false, depending upon the machine in question. In configuring Cfengine, one qualifies each action with the conditions under which it should occur, expressed as a boolean expression of classes. For example, in the Cfengine code:

```
links:
  solaris.victim::
    /etc/sendmail.cf
      ->! mail/sendmail.cf
```

the link is only made if the classes solaris and victim are both true ('.' represents logical 'and').

Classes in Cfengine correspond roughly with Prolog facts of arity 0. Facts are present (and thus 'true') if they depict the current operating environment. When Cfengine starts executing, it discovers as many facts as it can about the system upon which it is executing. For a machine hillary, running Solaris 5.7, these include classes equivalent to the Prolog facts:

```
hillary.
solaris.
sunos_5_7.
```

as well as much other information about the machine and operating environment.

Cfengine variable assignments, such as

```
groups:
 victim = ( bill hillary monica )
```

appear to set victim to a string of names, but in actuality only determine whether the class victim is true or false. The class victim is true if one of the facts in the parens is true, false otherwise. These are not really assignment statements at all, but represent the Prolog rules:

```
victim:-bill.
victim:-hillary.
victim:-monica.
```

This says simply that anywhere victim appears, it is true if bill, hillary, or monica is true. Each of these is in turn true only if it represents the current machine name.

This simplicity gives Cfengine incredible speed, as it never needs to deal with classes with values other than true or false: true if a fact is true in this environment, false if not. Prolog is slower, but as a result of this slowness, gains the ability to customize administrative process far beyond Cfengine's capabilities.

Cfengine class qualifications correspond with qualification goals in Prolog rules, where all qualifications have arity 0. For example, the Cfengine rule:

```
links:
  solaris.victim::
    /etc/sendmail.cf
      ->! mail/sendmail.cf
    /etc/services
      ->! inet/services
```

(which makes a link from /etc/sendmail.cf to mail/sendmail.cf) corresponds with the Prolog rule:

```
links:-solaris,victim,
       link('mail/sendmail.cf',
            '/etc/sendmail.cf').
links:-solaris,victim,
       link('inet/services',
            '/etc/services').
```

Roughly translated: "If you want to do links, and you're executing under Solaris, and you're a victim, do these."

The control: section of Cfengine's configuration file specifies the sequence in which individual goals are assured. For example, the Cfengine configuration:

```
control:
  actionsequence = ( links copy )
```

(which says to do only symlinks and file copies, in that order) corresponds roughly to the Prolog code:

```
health:-links,fail.
health:-copy,fail.
?- health,fail.
```

We create an artificial goal health that represents the health of the whole computer. To assure health, we look at all aspects, including links and copy. The fail directives assure that we check all possible rules for each of these through backtracking.

Genericity

Cfengine has been a great inspiration to us, and is a crucial element in day-to-day operation of our site, but its limitations forced us to look elsewhere for a truly general-purpose language for configuring systems. Cfengine is fantastic for manipulating files, but is very difficult to use to create generic, platform-independent, reusable configuration instructions for implementing high-level services.

Using any tool, any time a system file can vary in location or format, one must construct a new special case rule or macro to handle the deviations. In Cfengine this process becomes unwieldy very quickly, as macros in Cfengine all inhabit one name space, and one must remember the meanings of all of them in order to write new rules.

For example, let us learn to deal with an inetd.conf file that moves between /etc/inetd.conf and /etc/inet/inetd.conf depending upon operating system. One can cope with this in Cfengine as follows:

```
editfiles:
 ftp.solaris::
  { /etc/inet/inetd.conf
    AppendIfNoSuchLine \
     "ftp stream tcp nowait root \
     /usr/sbin/in.ftpd in.ftpd"
  }
 ftp.osf::
  { /etc/inetd.conf
    AppendIfNoSuchLine \
    "ftp stream tcp nowait root \
     /usr/sbin/ftpd ftpd"
  }
```

To avoid unwieldy typesetting, we take some liberties with Cfengine examples; the \ is *not* recognized by Cfengine as a line break.

Using Cfengine macros, it is possible to code the same operation somewhat more neatly:

```
control:
 solaris::
  inetd = ( "/etc/inet/inetd.conf" )
  ftpd = ( "/usr/sbin/in.ftpd" )
  ftpd_base = ( "in.ftpd" )
 osf::
  inetd = ( "/etc/inetd.conf" )
  ftpd = ( "/usr/sbin/ftpd" )
  ftpd_base = ( "ftpd" )
editfiles:
 ftp::
  { $(inetd)
    AppendIfNoSuchLine \
     "ftp stream tcp nowait root \
     $(ftpd) $(ftpd_base)"
  }
```

The variables $(inetd), $(ftpd), and $(ftpd_base) represent varying quantities in an otherwise unvarying script.

This works fine, but has two significant drawbacks. These variables are macros, created by hand, and one cannot write a script to discover their values. Variables live in a flat name space, so that repeating this process leads to many variables and much to remember when writing configuration entries.

Using Prolog, one can accomplish the same task somewhat more neatly. First, we code a relation config_path(Name,OS,Path) into Prolog that relates the canonical name of a file with its location in the filesystem. This relation might start, e.g., with the tuples:

```
config_path(
  'inetd.conf', osf,
  '/etc/inetd.conf').
config_path(
  'inetd.conf', solaris,
  '/etc/inet/inetd.conf').
config_path(
  ftpd, osf,
  '/usr/sbin/ftpd').
config_path(
  ftpd, solaris,
  '/usr/sbin/in.ftpd').
```

This information concerns the nature of operating systems themselves, not their configuration, so that it *does not change with policies* and *should not vary with use*. By nature, thus, this information is fundamentally different than configuration information and should *not be present* in your policy description.

Then, using a functor os/1 that returns the generic name of the current operating system, one can write the rule:

```
editfiles:-
  os(Os),
  config_path('inetd.conf',Os,Path),
  config_path('ftpd',Os,Ftpd),
  file_base_name(Ftpd,FBase),
  appendIfNoSuchLine(Path,
    [ftp,stream,tcp,nowait,
     root,Ftpd,Fbase]).
```

In English, "If we know our operating system, and can determine the path of inetd.conf and ftpd, and can find the base name of ftpd, and can put a record into

inetd.conf for it, we're done!" This series of goals queries for the correct location for inetd.conf and ftpd, computes the base name of the file from the ftpd location, dynamically constructs a line for the file by concatenation, and places that line into inetd.conf (after constructing the line from a Prolog list).

This does the exact same thing as the Cfengine example, but the Prolog code can be modified to do considerably more. Consider the rules in Listing 1. These rules compute the paths for these files by *actively probing the filesystem for their existence* This covers all cases *without* having to hand-code the locations for each operating system, using tests similar to those used by configure and autoconf.

Now let us go to an even higher level of abstraction: we should only do this when ftp is required. First, let's only use the rule when we need that service, by adding a 'guard clause' to the Prolog code:

```
editfiles:-
  service(ftp),
  os(Os),
  config_path('inetd.conf',Os,Path),
  config_path('ftpd',Os,Ftpd),
  file_base_name(Ftpd,FBase),
  appendIfNoSuchLine(Path,
    [ftp,stream,tcp,nowait,root,
                    Ftpd,Fbase]).
```

Then we write rules telling when a particular machine deserves the service:

```
service(ftp):-hostname(monica).
service(ftp):-os(osf).
```

This means that we should provide that service if our hostname is monica or our operating system is OSF! We have the *full power of Prolog* available for deciding which machines get the service. We could, e.g., write:

```
service(ftp):-not passwd(bgates,
                         _,_,_,_,_,_).
```

to install ftp only on machines where Mr. Bill does not have an account!

The rule that actually adds the appropriate line to inetd.conf is *not* part of operating policy. It is a *reusable method* that works in most cases. The actual policy is embodied, instead, by the rules for service/1. Ideally, we should be able to write the former once and never touch it again, then modify the latter to taste for each site and application.

This represents a major difference between using Prolog and other languages for configuration. Typically, when one gets to a high-enough lingual level to describe policy, low-level details (such as your user database!) become inaccessible. Prolog allows one to craft high-level, service-based policies that utilize data from all facets of the running system.

Configuring High-level Services

While Cfengine does a good job of implementing policies regarding individual files, it is quite awkward to describe how to implement high level services using Cfengine's syntax. Take, for example, the case of setting up an entire FTP server. Everyone knows that there are at least three actions involved in setting up a typical server within a UNIX-like operating system:

1. Add an appropriate line to inetd.conf.
2. Add appropriate port descriptions to services
3. Send a HUP signal to inetd.

In Cfengine, to define ftp on three machines bill, andhillary, monica, a mix of Solaris and OSF machines, these actions could be declared as follows:

```
control:
 solaris::
  inetd = ( "/etc/inet/inetd.conf" )
  ftpd = ( "/usr/sbin/in.ftpd" )
  ftpd_base = ( "in.ftpd" )
  services = ( "/etc/inet/services" )
 osf::
```

```
config_path(
 'inetd.conf', _, '/etc/inetd.conf'):-
  path_type('/etc/inet/inetd.conf',file),!.
config_path(
 'inetd.conf', _, '/etc/inetd.conf'):-
  path_type('/etc/inetd.conf',file),!.
config_path(
 ftpd, _, '/usr/sbin/ftpd'):-
  path_type('/usr/sbin/ftpd',file),!.
config_path(
  ftpd, _, '/usr/sbin/in.ftpd'):-
  path_type('/usr/sbin/in.ftpd',file),!.
config_path(
  ftpd, _, '/usr/etc/in.ftpd'):-
  path_type('/usr/etc/in.ftpd',file),!.
```

Listing 1: Rules to probe filesystem actively.

```
   inetd = ( "/etc/inetd.conf" )
   ftpd = ( "/usr/sbin/ftpd" )
   ftpd_base = ( "ftpd" )
   services = ( "/etc/services" )
groups:
 ftp = ( bill hillary monica )
editfiles:
 ftp::
 { $(inetd)
  AppendIfNoSuchLine \
    "ftp stream tcp nowait root \
     $(ftpd) $(ftpd_base)"
 }
 { $(services)
  AppendIfNoSuchLine "ftp-data 20/tcp"
  AppendIfNoSuchLine "ftp 21/tcp"
 }
processes:
 ftp::
  "inetd" signal=hup
```

The control section describes locations of files for each platform. The groups section defines hosts that need to provide ftp as a 'logical macro' that is true if the current host is a member, false if not. The editfiles section tells what to do to inetd.conf and services for these hosts. Within this, there are variations depending upon whether the operating system for the host is Solaris or OSF. Finally, the processes section describes what to do to running processes, i.e., send a HUP signal to inetd.

This approach has several advantages. Each action is done at most once, even if needed in several cases, e.g., inetd is only HUP'd once even if several changes are made to inetd.conf. Once ftp configuration is described for each operating system, one need only list the machines that should support it.

But there are several problems with this approach from our perspective. It is difficult to specify parameters that modify implementation, e.g., using TCP wrappers for security [25], passing parameters to daemons, etc. Each variant requires that a new class and/or macro be created, and these exist in a global name space. Actions of each type are done 'all together' with no concept of installing 'one service at a time'. Thus there is no concept of transaction integrity or transaction rollback if necessary for any reason. Variables and classes have global scope, so in every section of the file, we must remember that solaris represents an operating system, while hillary represents a machine name, and we cannot name a machine solaris without serious problems!

How does the same complex example look in Prolog? First, let's list the hosts that should have ftp service in Prolog rules:

```
service(ftp):-hostname(bill).
service(ftp):-hostname(hillary).
service(ftp):-hostname(monica).
```

We add code to the above example specifying where extra files are:

```
config_path('services',solaris,
            '/etc/inet/services').
config_path('services',osf,
            '/etc/services').
```

Then we add a list of goals that implement ftp service, in the manner of the above:

```
ftp:-
 service(ftp),
 os(Os),
 config_path('inetd.conf',Os,Inetd),
 config_path('ftpd',Os,Ftpd),
 config_path('services',Os,Services),
 file_base_name(Ftpd,FBase),
 appendIfNoSuchLine(Inetd,
  [ftp,stream,tcp,nowait,root,
   Ftpd,Fbase]),
 appendIfNoSuchLine(Services,
  ['ftp-data','20/tcp']),
 appendIfNoSuchLine(Services,
  ['ftp','21/tcp']),
 kill(inetd,hup).
```

In English,
"If we have a host name;
and we know our operating system;
and we know where inetd.conf, services, and ftpd live;
and we can put a record into inetd.conf;
and we can put two records into services;
and we can send a hup to inetd;
then ftp is installed."
Each clause guards against poor installation, by making constant 'sanity checks' and aborting if any one check fails.

In this example, we have done something that is very difficult to accomplish in any current configuration tool. The actual script that implements the ftp service is *generic* and *independent of architecture*. It will work for any host provided that file location tables are kept up to date. Customization is only required for service(ftp), which must be changed to reflect current policies and desires.

We chose in the prototype to describe services and system attributes very differently from in the way they are described in Cfengine. The class hillary in Cfengine is the goal hostname(hillary) in Prolog, while Cfengine's fact solaris becomes os(solaris). Thus facts are no longer filed in a flat name space, and we can distinguish between solaris the operating system and solaris the host name, if any. This extra work sidesteps inherent ambiguities in the meaning of Cfengine's class names.

Atomicity and Rollback

Another significant cost to Cfengine's remarkable efficiency is that there is no provision for recovery from partial configuration failures. Let's craft a Prolog example that undoes a configuration if any part of it fails. This will have the effect of making the installation more of an *atomic* act, one indivisible

thing in which several changes are coordinated to achieve one effect. If any change fails, a *rollback* script will undo the other changes so that system integrity is maintained.

This example will contain two goals for ftp, one that installs it and one that removes it if anything goes wrong. First, we add a Prolog cut (!) as the last goal in the above installation script. This tells Prolog that when all subgoals are complete, the ftp goal itself is complete and no further work should be done on it. We then follow this rule with another that only gets executed if the cut is *not* encountered:

```
ftp:-
  service(ftp),
  os(Os),
  path('inetd.conf',Os,Inetd),
  path('ftpd',Os,Ftpd),
  path('services',Os,Services),
  file_base_name(Ftpd,FBase),
  deleteLinesContaining(Inetd,Ftpd),
  deleteLinesContaining(Services,
                        '20/tcp'),
  deleteLinesContaining(Services,
                        '21/tcp'),
  kill(inetd,hup).
```

When Prolog tries to do something, it tries every relevant rule in its database of rules, in the order in which they appear in its program. When asked to handle the rule for ftp, Prolog will begin by trying the initial rule we crafted. If that rule succeeds, then because of the ! (cut) at the end, Prolog will stop working on that goal. If the initial rule fails, it will try the next rule, which uninstalls ftp service.

This example shows the true power of implicit goal execution. As a script, this behavior would be a nightmare to describe, but in Prolog, it is a simple series of two rules, each of which is tried if the last one fails. Thus a rather complex logical chain of deduction is reduced to a relatively simple list of requirements.

Cfengine and Dynamic Policy

Cfengine only implements dynamic policy where the map from policy to process is relatively obvious, or the user is willing to allow Cfengine to make arbitrary decisions concerning the mapping. For example, if a user has temporary files that are too old, most everyone agrees that the obvious thing to do is to delete them, and Cfengine can do this easily. If, however, one wishes to archive them on tape or writeable CDROM, Cfengine cannot help very much, and one must write a custom script.

As a worst-case example, it is not so obvious that everyone should use the NFS disk management strategy imposed by Cfengine simply because Cfengine supports only that strategy. Unless one conforms somewhat precisely to a rather elaborate scheme, including a naming convention for network directories containing the name of the server, several features of Cfengine are unavailable. As we feel that mount points should be machine-independent (from bitter experience in moving user files and having to repair user scripts), we choose not to utilize these Cfengine features. Cfengine tries to impose a rather significant operating policy decision upon us – one we cannot afford to allow.

In implementing most dynamic policies, the map from policy to convergent process is so ill-defined that it becomes a policy decision itself. For example, at our site, users can gain access to a 'temporary storage' area that has no quota, for the purpose of doing things that require more storage than will fit into their home directories. But we would like to impose a time limit on peoples' use of that storage that is different from a normal quota. Suppose we find out that a user is using a large amount of temporary storage for too long. How do we 'correct' that state? We could:

1. Delete some files randomly from temporary storage and mail a message.
2. Mail a warning, wait a week, then lock the account until the user comes by to talk about the problem.
3. Invoke a temporary quota on the temporary storage area for this user, to force the user to clean up.
4. Write email to a system administrator describing the problem.

Clearly, the option we choose determines much about how users work and feel.

Prolog and PIKT

We chose Prolog as our prototyping language partially because of the intimate relationship between it and Cfengine. But we also wished to be able to control dynamic state, including manipulating user files, filesystems, and processes. The powers and ease of use of PIKT [24] inspired us to attempt to add those powers to the prototype without compromising Cfengine-like behavior.

There are remarkable similarities between Cfengine and PIKT. Both implement roughly the same idea of classes, but to slightly different ends. In Cfengine, a class is a guard mechanism that determines which rules apply. In PIKT, a class instead determines which lines are used in a script. In both, classes are primarily a portability mechanism. In Cfengine, classes insulate one from differences in file layout and location, while in PIKT, classes determine which scripts apply to which operating systems, and help one cope with differences in command output formats between operating systems. Cfengine's classes are logical variables, while in PIKT, classes are variables in the C preprocessor that become defined or undefined for each platform. Portability in PIKT's scripts is accomplished much like portability in C programs, by enclosing variants in preprocessor #if... #endif directives.

Like Cfengine, PIKT's configuration is separated into several distinct parts. While Cfengine operates on files, links, and processes, PIKT acts to detect alarm

conditions and perform appropriate actions. In understanding how we can implement PIKT functions in Prolog, there are four parts to consider: classes, macros, alarms, and actions.

PIKT classes are specified much as in Cfengine. The class of three machines bill, hillary, and monica (from before) might be constructed as:

```
watch
    members bill hillary monica
```

However, PIKT interprets classes only on a master script server, not on the target machine being configured. This server uses class definitions and preprocessor directives to adapt generic master scripts to execute on the target machine, and then ships a class-free script to the target machine for actual execution.

The resulting scripts are very efficient, because many decisions have been made before shipping the script to the target machine. However, this also means that PIKT scripts cannot rely on any form of knowledge discovery in creating classes, as in Cfengine and Prolog. Classes that Cfengine and Prolog can discover automatically must be explicitly declared by hand in PIKT, including machine type and operating system version.

In PIKT, as in Cfengine, macros can be used to code file locations and other local dependencies. For example, in the PIKT file macros.cfg, one might write:

```
#if solaris
fstab     /etc/vfstab
#endif
#if osf
fstab     /etc/fstab
#endif
```

This makes the macro =fstab evaluate to /etc/vfstab when executing within Solaris and /etc/fstab within OSF.

Alarms in PIKT are specified by listing, for each host, a set of scripts to be run periodically to check for system problems. Each script is configured to execute regularly and take action if needed. This feature cannot be emulated by our prototype, but the Prolog interpreter can always be run periodically under control of the cron periodic execution daemon.

Scripts in PIKT operate on variables read from logfiles and the output of UNIX commands. For example, the script:

```
AnnoyBill
  init
    status active
    level critical
    task "Harass Bill"
    input proc "=w | =grep clinton"
    dat $tty 2
  rule
    exec wait =write \
          clinton $tty < =mesg
```

will check periodically whether Mr. Bill is logged in, and write an undisclosed message to each terminal on which he is working!

The init section describes conditions under which to do something. The input statement describes a filter that only generates input to which the rule should be applied. In this case, the rule will be applied whenever any line in the output of the w command contains the string clinton. The second field of that line (the tty field) will be assigned to the variable $tty before invoking the rule. The rule will call the program write to send a message to that tty, where the macros =write and =mesg must describe the locations of the write command and message file, respectively.

It is not surprising that such a rule-based execution is very easy to accomplish in Prolog. In our prototype, an equivalent rule looks like this:

```
annoyBill:-
  os(Os),
  command_path(w,Os,WPath),
  output_tail(WPath,1,Out),
  split(Out,'[ \t][ \t]*',[clinton,Tty|_])
  command_path(write,Os,WritePath),
  file_path(message,Os,MessPath),
  concat_atom(
    [WritePath,clinton,Tty,
       '<',MessPath],' ',Command),
  system(Command).
?- annoyBill,fail.
```

The command_path/3 and file_path/3 goals compute where needed commands and files live. output_tail/3 executes a command and then binds Out successively to each line after the first during backtracking. split/3 splits this line into fields at spaces, assigning the second field to Tty only if the first field is clinton. When this happens, the write command is built and executed.

Because each Prolog subgoal acts as a natural filter that limits further operations to valid data, Prolog easily emulates the function of the PIKT script while adding additional safeguards against erroneous operation. The PIKT script will misbehave if one forgets to define =w, for example, while the Prolog rule will stop executing in that case. Admittedly, this is much less efficient that computing system dependencies before running the script, as PIKT does, but PIKT functions can be emulated with some performance loss.

Unlike PIKT, however, it is easy to write this rule 'at a higher level of abstraction,' by hiding system dependencies in subgoals. Consider the rule:

```
w(User,Tty):-
  os(Os),
  command_path(w,Os,WPath),
  output_tail(WPath,1,Out),
  split(Out,'[ \t][ \t]*',[User,Tty|_]).
```

This rule tells how to execute a w command and present the results through backtracking. Once this is written, the above script can be written:

```
annoyBill:-
  w(clinton,Tty),
  os(Os),
  command_path(write,Os,WritePath),
  file_path(message,Os,MessPath),
  concat_atom(
    [WritePath,clinton,Tty,'<',
            MessPath],' ',Command),
  system(Command).
```

Just like writing a subroutine in a script, writing the w/2 subgoal allows one to forget about the details of running the w command and concentrate on the action to take. One can do the same with the action of writing the message to make the rule even more readable:

```
annoyBill:-
  w(clinton,Tty),
  file_path(message,Os,MessPath),
  write(clinton,Tty,MessPath).
```

The Prototype

Our prototype Prolog system administration interface is based upon SWI-Prolog 3.7.2 [27], a freely available interpreter that executes both under UNIX and NT. Its many features include built-in functions for manipulating files and the ability to call dynamically loaded C functions from within Prolog programs. This made it easy to adapt the language for administrative tasks.

We explored the potential of using logic programming for configuration control by writing rather simple 'interface' rules, like the ones in the above examples, that expose system configuration and dynamic state, and perform common actions. Then we tried to do the same things with Prolog that we would do with normal tools and scripts.

The prototype's system interface began as a very simple hack. When a kind of system fact was needed, the Prolog program executed a Perl utility called "glue." This utility queried the system and provided the results as Prolog facts. Whenever any fact of a certain kind was needed, all such facts were loaded, to minimize external system calls and avoid program execution overhead. This was a very rough approximation to what should really be done, and with great effort, we converted the prototype to use the shared library support built into the SWI-Prolog interpreter. This allows information to be transferred directly from system calls into the interpreter with no script execution overhead. The current prototype can access system information as quickly as a C program using the same system calls.

Alas, this is only a prototype and has several serious limitations. The prototype only compiles under Sun Solaris (which the bulk of our hosts utilize), and little effort has been made to port it to other architectures. There is no provision for file transfer between hosts, and file editing is extremely primitive by Cfengine standards. The interface implements very few of the capabilities built into Cfengine: just enough to perform some convincing tests, as above. Results of these tests, however, strongly encourage us to implement more features as time allows.

Performance

It may seem, from the preceeding examples, that programming in Prolog is easy. This is false! Our prototype simply hides the details of real programming from the administrator, by providing predefined rules one can use. These predefined rules are actually quite complex to craft with any kind of efficiency.

Prolog excels at implying complexity from form. Usually, a simple statement of what should happen is enough to make it happen: Prolog infers the process to do this from the needs one describes. It is very difficult, however, to make Prolog do something efficiently, because there are many valid descriptions of how to achieve the same effects, with great variations in performance.

In any high-level approach, we trade specificity and control for ease of use. One of the reasons Prolog is so attractive to us is that describing an instance suffices to operate on all instances. Because of this power, however, we lose the ability to easily control iteration in the way to which we are accustomed when writing scripts. In particular, there is no easy way to craft a nested loop that iterates over all pairs in the same set.

Here is a real life example of something difficult to do in Prolog. We have a directory full of electronic submissions of homework that we would like to check for similarity. To do this, we would like to run diff on all pairs and locate pairs with few differences. We could write:

```
diff:-
  expand_file_name('*',Nodes),
  member(File1, Nodes),
  member(File2, Nodes),
  concat_atom_chars(['diff',File1,
                File2],' ',Command),
  system(Command).
?- diff,fail.
```

In this code, we first obtain a Prolog list of all files in the current directory, then select File1 and File2 from this list and compare them. It is not so obvious that this does twice as much work as necessary, because the two implied loops for selecting files not only compare files against themselves, but also against all other files in both orders. This does more than twice as many comparisons as we need.

The most efficient implementation of this loop is:

```
pair([File1|Rest],File1,File2):-
  member(File2,Rest).
pair([_|Rest],File1,File2):-
  pair(Rest,File1,File2).
```

```
diff:-
  expand_file_name('*',Nodes),
  pair(Nodes,File1,File2),
  concat_atom_chars(
    ['diff',File1,File2],
    ' ',Command),
  system(Command).
?- diff,fail.
```

In English,
"To diff all files,
get all filenames in a list;
select pairs from that list;
and diff them."
The complexity here is in pair/3: "To select a pair, make it the first element of the list along with some other, or repeat that process with some suffix of the original list." Prolog efficiency is *not* an oxymoron, but it takes an expert Prolog programmer to consistently generate non-trivial and efficient Prolog code.

Conclusions

In no way is our prototype the equal of Cfengine or PIKT, but it does have very important capabilities not present in either. Using the prototype, one can concentrate on what should happen in each case, and leave scripting of the actual changes to Prolog itself. With a few more relatively simple extensions, Prolog can in principle accomplish anything that either of these tools can do. Prolog programs are not subject to either the assumptions built into Cfengine or the lack of dynamic probing capabilities in PIKT. Prolog scripts are roughly the same length as scripts in either Cfengine or PIKT, but much easier for an expert Prolog programmer to refine for readability and extend for new capabilities. The chain of logical deduction involved in Prolog execution is a close match with the way configuration processes should work. Goals to be used in that chain can be crafted to force the system into compliance with requirements, or to actively probe for system problems. Custom scripts to accomplish special purposes can be written in Prolog without recourse to external scripting languages.

The basic programming metaphor for Prolog, *unification*, matches exactly what we have to do in creating a convergent process: to *unify* the rules with the system so that both "describe the same thing." This is not an easy task in any sense, but the fact that the language in some sense "matches the problem" makes crafting complex processes easier than when using normal scripting languages or less flexible configuration languages.

Our prototype illustrates several important lessons about the problem of configuration and the power of language. It is possible to configure a system using a language in which describing a single instance of a problem suffices to repair all instances. It is possible to craft service installation scripts that are truly generic, so that the administrator need not program, but simply correctly populate databases describing the system and desired behavior. It is possible to separate data concerning the invariant system from data describing operating policies. It is possible to describe invariant system data once and reuse it for all similar cases. It is possible to describe both static and dynamic configuration issues with a single language.

The prototype also reiterates lessons we have learned whenever we attempt to operate 'at a higher level' than existing tools. Simplicity of a language reduces the ability to craft efficient programs. Language flexibility increases the potential for confusion in reading programs. Undisciplined use of a flexible language leads to unpredictable results.

We do *not* suggest that every administrator should learn to program in Prolog! Prolog programs are difficult to write correctly and efficiently. Even in this simple prototype, one must often repeat code in several rules in order to emulate classes utilized in one or two lines of Cfengine or PIKT configuration. We do not view Prolog as a language for administrators to use directly, but as an *assembly language* into which even higher level descriptions can eventually be compiled. This common language for both static and dynamic configuration management, though cumbersome in its raw form, can be made much more friendly by some relatively simple syntactic translations.

Ideally, a true Prolog configuration tool would handle *all* the low level details and portability issues, leaving us to decide the high-level policies to implement. File and command location databases could be built with configure, to be used in generic implementation routines as needed. Configuring the system would consist of deciding which services to offer and which periodic configuration tests to enable. Custom Prolog programming would only be needed if an administrator wished to extend the tool's capabilities by adding new services or tests. Our prototype is nowhere near this ideal, but shows us that this ideal is possible to attain with effort.

Future Work

Our work on the prototype has only just begun. We have a long 'wish-list' of features to add, all of which are relatively straightforward to implement. This list includes giving the Prolog interpreter:

1. generic interfaces to SQL, LDAP, and NIS+ database services, both to request and modify data.
2. domain-specific interfaces to system files, both for scanning and updating file contents.
3. extensive, Cfengine-like file editing capabilities.
4. a generic interface to the Simple Network Management Protocol (SNMP) [20].
5. the ability to request master files from Cfengine configuration servers.

We are also involved in writing a preprocessor that will translate easier-to-use configuration instructions into Prolog, avoiding the need for anyone but the system designer to code in Prolog directly.

A Simple Dream

Everyone working in configuration management would like to find a way to simplify and perhaps obsolete this dreary job. The 'impossible dream' is that everyone writing configuration scripts can use the work of the whole community instead of re-inventing the wheel for each site and purpose. But so far, while configuration tools proliferate, it has been difficult to convince people to distribute and maintain reusable scripts for configuring systems and implementing common services. There seem to be "too many options," "too many system dependencies," and "too many site-specific assumptions."

Our work shows that using logic programming, one can break the problem of service installation into small steps so that the deliverables at each step will be maintainable. These steps can be coded in a single language with wide applicability. This language may not be Prolog, but all evidence suggests that it will have quite similar capabilities. We will know we have succeeded when 'unification' is something we can do to human effort as well as system behavior.

Acknowledgements

Many people inspired and otherwise contributed to this work. We are forever indebted to Mark Burgess, whose work on Cfengine showed us that our dreams were possible, and inspired us to look beyond the obvious for a better way. We thank Jan Weilemaker for providing SWI-Prolog, without which writing the prototype would have been difficult if not impossible. We especially thank him for timely and extremely helpful responses to our bug reports on SWI-Prolog and for putting up with our lack of understanding of how to write foreign extensions to Prolog. We thank Robert Osterlund for PIKT, and for showing us both the difficulty of dynamic monitoring, and the way to effectively handle script heterogeneity in a complex and varied environment. Finally, we thank the EECS systems staff, George Preble and Warren Gagosian, for being there when we needed them and keeping our systems running smoothly.

Author Information

Alva L. Couch was born in Winston-Salem, North Carolina where he attended the North Carolina School of the Arts as a high school major in bassoon and contrabassoon performance. He received an S.B. in Architecture from M.I.T. in 1978, after which he worked for four years as a systems analyst and administrator at Harvard Medical School. Returning to school, he received an M.S. in Mathematics from Tufts in 1987, and a Ph.D. in Mathematics from Tufts in 1988. He became a member of the faculty of Tufts Department of Computer Science in the fall of 1988, and is currently an Associate Professor of Electrical Engineering and Computer Science at Tufts. Prof. Couch is the author of several software systems for visualization and system administration, including Seecube (1987), Seeplex (1990), Slink (1996) and Distr (1997). In 1996 he also received the Leibner Award for excellence in teaching and advising from Tufts. He has assisted in maintaining the Tufts computer systems for Computer Science teaching and research since 1985, when he was a Ph.D. student. With a lot of help, Prof. Couch still maintains the largest independent departmental computer network at Tufts in the department of Electrical Engineering and Computer Science. He can be reached by surface mail at the Department of Electrical Engineering and Computer Science, 161 College Avenue, Tufts University, Medford, MA 02155. He can be reached via electronic mail as <couch@eecs.tufts.edu>. His work phone is (617)627-3674.

Michael Gilfix was born in Winnepeg, Canada on Aug. 3rd, 1980. He presently resides in Montreal, Canada, where he attended high school at Lower Canada College. He is currently a sophomore at Tufts University, double-majoring in Electrical Engineering and Computer Science. His many interests include Jazz and Rock music, playing improvisational guitar, movies, and shooting pool. He can be reached by electronic mail as <mgilfix@eecs.tufts.edu>.

References

[1] P. Anderson, "Towards a High-Level Machine Configuration System" *Proc. LISA-VIII*, Usenix Assoc., 1994.

[2] E. Bailey, *Maximum RPM*, Red Hat Press, 1997.

[3] M. Burgess, "A Site Configuration Engine," *Computing Systems* **8**, 1995.

[4] M. Burgess and R. Ralston, "Distributed Resource Administration Using Cfengine," *Software: Practice and Experience* **27**, 1997.

[5] M. Burgess, "Computer Immunology," *Proc. LISA-XII*, 1998.

[6] W. F. Clocksin and C. F. Mellish, *Programming in Prolog, Fourth Edition*, Springer-Verlag, Inc., 1994.

[7] Wallace Colyer and Walter Wong, "Depot: a Tool for Managing Software Environments," *Proc. LISA-VI*, Usenix Assoc., 1992.

[8] Michael Cooper, "Overhauling Rdist for the '90's," *Proc. LISA-VI*, Usenix Assoc., 1992.

[9] Alva Couch and Greg Owen, "Managing Large Software Repositories with SLINK," *Proc. SANS-95*, 1995.

[10] A. Couch, "SLINK: Simple, Effective Filesystem Maintenance Abstractions for Community-Based Administration," *Proc. LISA-X*, Usenix Assoc., 1996.

[11] A. Couch, "Chaos Out of Order: A Simple, Scalable File Distribution Facility for 'Intentionally Heterogeneous' Networks," *Proc. LISA-XI*, Usenix Assoc., 1997.

[12] C. J. Date, *An Introduction to Database Systems, Sixth Edition*, Addison-Wesley, Inc., 1995.

[13] R. Evard, "An Analysis of UNIX Machine Configuration," *Proc. LISA-XI*, Usenix Assoc., 1997.

[14] B. Glickstein, "GNU Stow," http://www.gnu.org/software/sto w.

[15] S. Hansen and T. Atkins, "Centralized System Monitoring With Swatch," *Proc. LISA-VII*, Usenix Assoc., 1993.

[16] G. Kim and E. Spafford, "Monitoring File System Integrity on UNIX Platforms," InfoSecurity News **4** (4), July 1993.

[17] G. Kim and E. Spafford, "Experiences with Trip-Wire: Using Integrity Checkers for Intrusion Detection," *Proc. System Administration, Networking, and Security-III*, Usenix Assoc., 1994.

[18] W. Ley, "LogSurfer Homepage," http://www.fwl.dfn.de/eng/logsurf/home.html .

[19] J. Lockard and J. Larke, "Synctree for Single Point Installation, Upgrades, and OS Patches," *Proc. LISA-XII*, Usenix Assoc., 1998.

[20] J. Murray, *Windows-NT SNMP: Simple Network Management Protocol*, O'Reilly and Assoc, 1997.

[21] K. Manheimer, B. Warsaw, S. Clark, and W. Rowe, "The Depot: a Framework for Sharing Software Installation Across Organizational and UNIX Platform Boundaries," *Proc. LISA-IV*, Usenix Assoc., 1990.

[22] A. Oram and S. Talbot, *Managing Projects with Make, 2nd Edition*, O'Reilly and Associates, 1991.

[23] J. Ousterhout, *TCL and the TK Toolkit*, Addison-Wesley, Inc., 1994.

[24] R. Osterlund, "PIKT Web Site," http://pikt.uchicago.edu/pikt ..

[25] W. Venema, "TCP WRAPPER, Network Monitoring, Access Control and Booby Traps," *Proc. UNIX Security Symposium III*, September 1992.

[26] L. Wall, T. Christiansen, and R. Schwartz, *Programming Perl, 2nd edition*, O'Reilly and Assoc., 1996.

[27] J. Weilemaker, "SWI Prolog Web Site," http://www.swi.psy.uva.nl/projects/SWI-Prolog .

[28] Walter C. Wong, "Local Disk Depot – Customizing the Software Environment" *Proc. LISA-VII*, Usenix Assoc., 1993.

NetReg: An Automated DHCP Registration System

Peter Valian and Todd K. Watson – Southwestern University

ABSTRACT

NetReg is an automated system that requires an unknown DHCP client to register their hardware before gaining full network access. Through a simple web interface, the client is prompted for their user identification. Powerful scripts then retrieve the client's network fingerprint and store it along with the user's information in a database. The database provides administrators with real-time information for troubleshooting and auditing their networks. The entire system was developed utilizing unmodified, open source servers and in-house developed programs.

Introduction

At Southwestern University, we want to provide an "open" network that does not limit student access and gives them an easy and instant connection. In the past, we have used a simple Dynamic Host Configuration Protocol (DHCP) [5] server to assign IP addresses to any client connecting to our residential network (ResNet). This system worked well; however, it did not provide system administrators adequate information about clients connected to the network. Tracing a network address to an end-user proved to be a tedious and time-consuming task due to the lack of a unified data source containing information about what hardware belongs to which user. We wanted to improve and automate this system while maintaining client-side simplicity.

A few commercial products exist which will perform various types of network registration (such as Cisco's *Network Registrar* and Lucent's *QIP*)[1]; however, these products are often expensive and provide functionality not necessarily required for a simple registration system. None of these products fit our needs 100%, so it didn't make sense to spend a lot of money on them. Some universities have created registration systems which provide the functionality we were looking for, but they lacked the simplicity of using off-the-shelf servers without any modifications. This was mostly a result of DHCP servers not having needed features (such as supporting multiple address pools per subnet) when they developed their registration systems. Time and programming staff is limited at our small liberal arts school, so we needed something that would work well without having to put in a lot of development work.

The release of version 3 of the Internet Software Consortium (ISC) DHCP server[2] brought a free server supporting multiple address pools per subnet. This was what we needed to create our system based on readily available open-source servers without modifications.

The Registration System

The first step in designing our system was to outline what we wanted as an end product. We needed to have a single source of information linking a Media Access Control (MAC) address or IP address to a specific user ID. Each user on our campus network is responsible for his/her computer and university computer accounts as specified in our Acceptable Computer usage policy. We wanted an "electronic signature" stating that the user has read and understands the usage policy before gaining access to the network. We also needed a means of denying access to the network without proper credentials. In addition to functionality, we wanted a nice front-end administration interface that would not require the knowledge of command-line parameters or file names. The interface also needed to be cross-platform so Unix, PC, and Macintosh support people could administer the registration data.

The administration interface needed to provide basic search capabilities such as finding the user ID that registered a particular MAC address or finding out what MAC address is using a certain IP address. Also, NetReg needed to be powerful enough to cross-reference this information for queries such as, "Tell me what is the last user to use this IP address."

The Platform

Due to the way we decided to use DNS, we knew we needed to have a dedicated machine for NetReg. In the interest of our limited budget (and because we're partial to penguins), we decided to use RedHat Linux on a retired 200MHz Pentium. Since our residential network is so small (roughly 750 nodes), a more powerful machine wasn't necessary[3]. A basic installation

[1]These seem to be the two most commonly used commercial systems at least in the university arena.

[2]At the time of publication, the current release was 3.0b1pl0. We have been running version 3 since the initial Alpha version, and it has been extremely stable.

[3]Actually the system runs at such a low load average that it should be scalable to thousands of nodes without problem.

of the operating system was needed – no kernel hacks or windowing systems – just basic networking with a static IP address. On our Cisco router, we specified the IP of the NetReg machine as the **ip helper-address** in each subnet declaration of our residential network.

The Servers

Three servers are needed to make NetReg work: DHCP, DNS/BIND and HTTP. Again, our goal was to use reliable and low-cost software. We were very pleased that we managed to use completely open-source, well-known, heavily tested and hence very reliable servers. RedHat Linux 6.0 comes with BIND 8 and Apache 1.34. We chose to upgrade to the most recent version of Apache[4], but used the BIND server included with RedHat 6.0. Alternatively we could have used a simple Perl (or similar) program to act as a DNS server instead of the full DNS/BIND package, but since it is bundled with Linux, is well-documented, and does what we need, we didn't look any further. The ISC DHCP server is open-source; however, they have recently begun offering support contracts for those wanting commercial support. Similar contracts are available for RedHat (which support the servers shipped with it).

DHCP

The ISC's DHCP server version 3 was a dream come true. We consider this package to be most crucial to making NetReg possible because of its multiple pools per subnet capability. The first step was to define two IP address pools per subnet for each of our residential network subnets (see Figure 1).

```
subnet 161.13.1.0 netmask 255.255.255.0 {
  option routers 161.13.1.1;
  # Unknown clients get this pool.
  pool {
    option domain-name-servers 161.13.1.25;
    max-lease-time 120;
    default-lease-time 120;
    range 161.13.1.100 161.13.1.150;
    allow unknown clients;
  }
  # Known clients get this pool.
  pool {
    option domain-name-servers 161.13.1.2;
    max-lease-time 28800;
    default-lease-time 28800;
    range 161.13.1.151 161.13.1.250;
    deny unknown clients;
  }
}
```

Figure 1: Defining Multiple Pools per Subnet in dhcpd.conf

One pool would be small and only allow unknown clients to get a DHCP lease for a short amount of time (two or three minutes), and the other pool would be larger and grant leases to known clients for longer periods of time (24 hours). Known clients are those which have their MAC address defined in the server's configuration file (more on how a client becomes "known" will be explained in following sections). Unknown clients will receive a temporary IP address and the NetReg machine IP as their DNS server. Known clients will receive an IP address and the true DNS server address. A variety of configurations can exist based on the network topology. The best method is to use temporary addresses which are not globally routable.

DNS/BIND

The NetReg machine also runs as a "fake root" DNS server. Essentially, the DNS server is told not to communicate with any real DNS servers, but to simply resolve all queries against its own database file instead. Its database file has a very simple, one-line wildcard configuration telling the server to resolve every address to the NetReg IP address. In doing this, clients that received the "unknown client" configuration from the DHCP server are using the restricted "fake root" DNS server to resolve all their DNS queries. Unknown clients are effectively locked out from accessing any machine except the NetReg machine[5]. Use of this will become clearer in following sections.

HTTP

One more server ties everything together – the HTTP server. For this server we used the ever-popular Apache HTTP daemon. A basic installation was needed with little special configuration. The main index HTML page is a simple form with an input for a user ID and password (see Figure 2). The only modification made to the configuration was to redirect errors to the main index URL. To do this, we simply added the following lines in Apache's httpd.conffile:

```
ErrorDocument 404 /
ErrorDocument 403 /
```

This will effectively redirect any page for any requested site to the registration page.

The Process

And now the answers to all your questions[6]. There is a graphical flowchart of the process we are about to describe in Figure 3. When a client connects to the network via DHCP, the router will forward the "DHCP Request" packet[7] to the DHCP server on the NetReg machine. The DHCP server on NetReg will determine if the client is "known" or "unknown" and grant it the appropriate IP and DNS server information. If the client is unknown, the DHCP server

[4]We upgraded to Apache 1.3.6, however, since then version 1.3.9 has been released.

[5]The users could technically work around this by using an IP address instead of a URL, but it becomes much of a hassle to browse the web like that!

[6]OK, maybe not **all** of your questions, but at least those pertaining to NetReg.

[7]DHCP is a broadcast protocol which is squelched at the gateway interface of most segmented networks. Most routers can be configured to forward boot requests to one or more IP addresses.

assigns the "fake root" DNS server which resolves everything to itself. So now, if an unknown client opens a web browser and attempts to access any URL (e.g., http://www.usenix.org/), the "fake root" DNS server will resolve the server name to the IP of itself. The HTTP server on NetReg will serve that request with its index HTML file, which is the registration page for unknown clients. Now, if an unknown client requests a particular file or directory (e.g., http://www.usenix.org/events/lisa99/) the server part of the URL will resolve to the NetReg machine, but it will still want a particular page or directory which will most likely not exist. This is where the error redirects come in. The 404 (Not Found) and 403 (Forbidden) errors get redirected to the main index HTML – which is, of course, the registration page!

On the registration page (see Figure 2), the user is with the full text of our Acceptable Network Usage Policy. The page explains that by clicking the "Accept" button they signify that they have read and understand the policy. If they do not wish to accept the policy, they cannot gain access to the network from their machine.

How It Works

When the user clicks "Accept", the form data are sent to a Perl CGI script. The first thing the script does is authenticate the user against our POP server via the **Mail::POP3Client** Perl module [3] (all users on our network use this server for their mail, so if they have an e-mail account, they are permitted to use our network). Once the user is authenticated, the script grabs the user's IP address from the environment

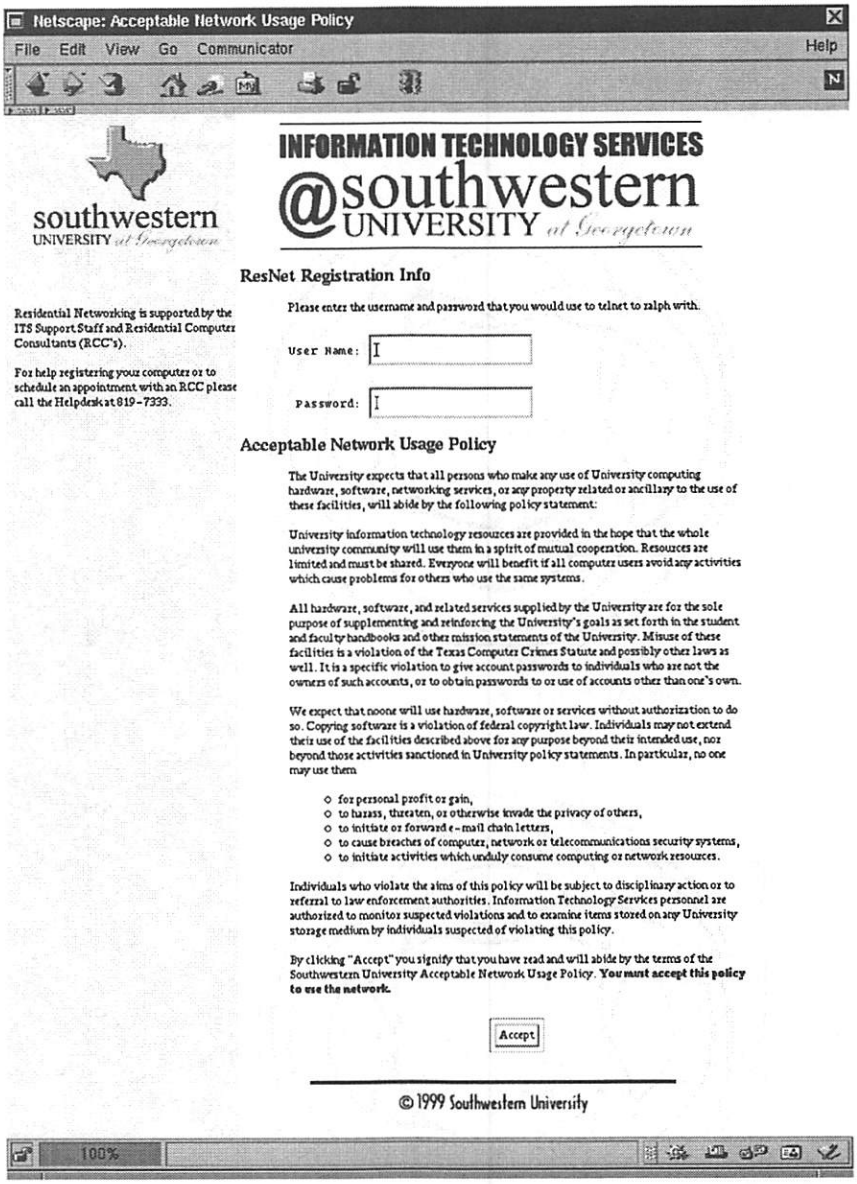


Figure 2: The NetReg Registration Page.

variables sent along with the CGI data. The user's IP address is searched for in the DHCP server leases database in order to obtain the user's MAC address. The user's MAC address is then added to the DHCP server configuration file along with other commented information about that MAC address, such as user ID, browser type used to register, time of registration and which subnet the user was connected to at time of registration.

Every minute, a script running from cron checks if there are any new registrations since the last time the DHCP server restarted. If there are, the cron script kills the server and starts it again so it reads the newly written configuration file[8]. During this time, the user is asked to reboot their system. Not only does this provide enough time for the DHCP server to reread its configuration, but also helps some clients with poor implementations of DHCP. We have found that sometimes a client will not change its DNS server to the real DNS server when it goes to renew its lease. Other times, we have found clients using both the fake and real DNS servers in a round-robin fashion. We opted to tell users to reboot rather than explaining how to release and renew a DHCP lease for each platform, while it gives the delay needed by cron[9] to reload the new dhcpd.conffile. Once registered, our DHCP server configuration will allow users full access to our campus intranet as well as the Internet. Users may also

[8]The DHCP server does not currently support HUP but may support on-the-fly configuration reloads in the future.

[9]Cron will only execute scripts once every 60 seconds at most.

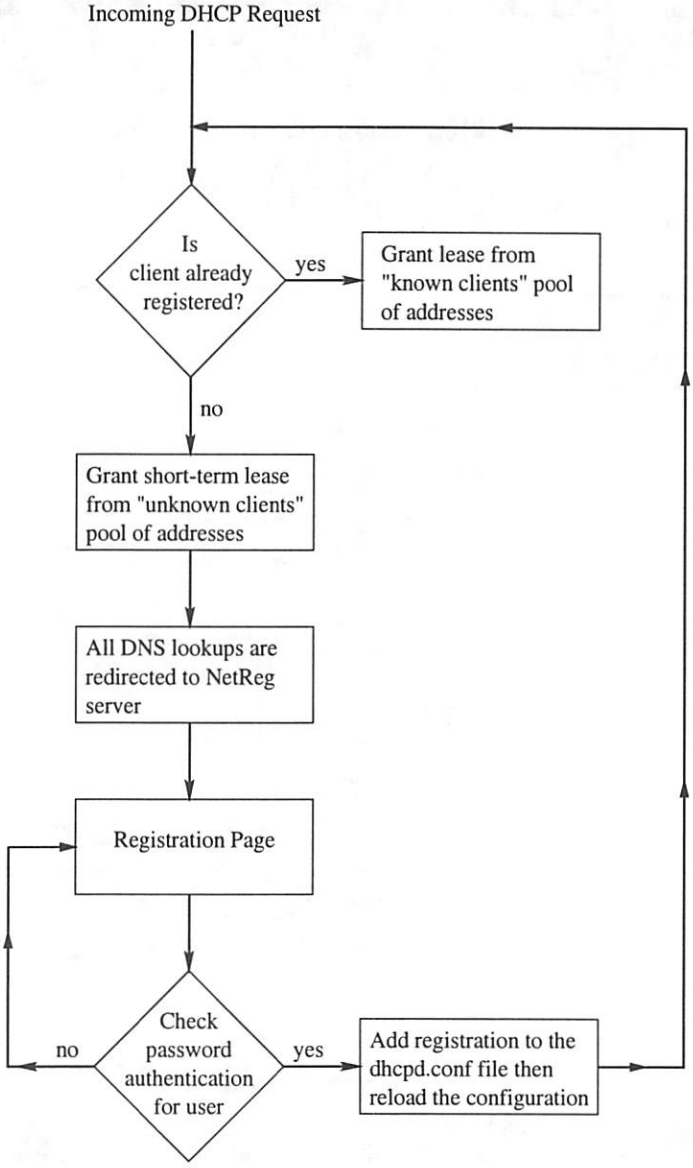


Figure 3: The NetReg Process

roam to any student lab or residential subnet and have the same network privileges as they would from their room without changing any network setting on their machine and without having to re-register.

Administrative Interface

NetReg is a great registration system, though the real usefulness comes in the administrative interface, which allows a network administrator to view, search, and manage the collected data. A single Perl CGI program provides an initial view of all subnets (see Figure 4) with a graphical representation of the number of clients registered on each subnet. From there, the admin can then view the individual registrations for a particular subnet (see Figure 5), perform a search for an indivual MAC address, user ID, IP address, or check the current status of the servers running on the NetReg system.

Although you can allow the administrative interface to be accessed through the same HTTP server as the registration page, we opted to set up a separate server to do this. For this server we used the SSL patched [6] version of Apache 1.3.6 using OpenSSL [7].

There are some nice features that are built in to the web-admin interface. For example, if we search for a MAC address we will then be able to query the system for the user ID that registered the MAC address. From there we can see all registered MAC addresses for that user ID, or we can perform a query

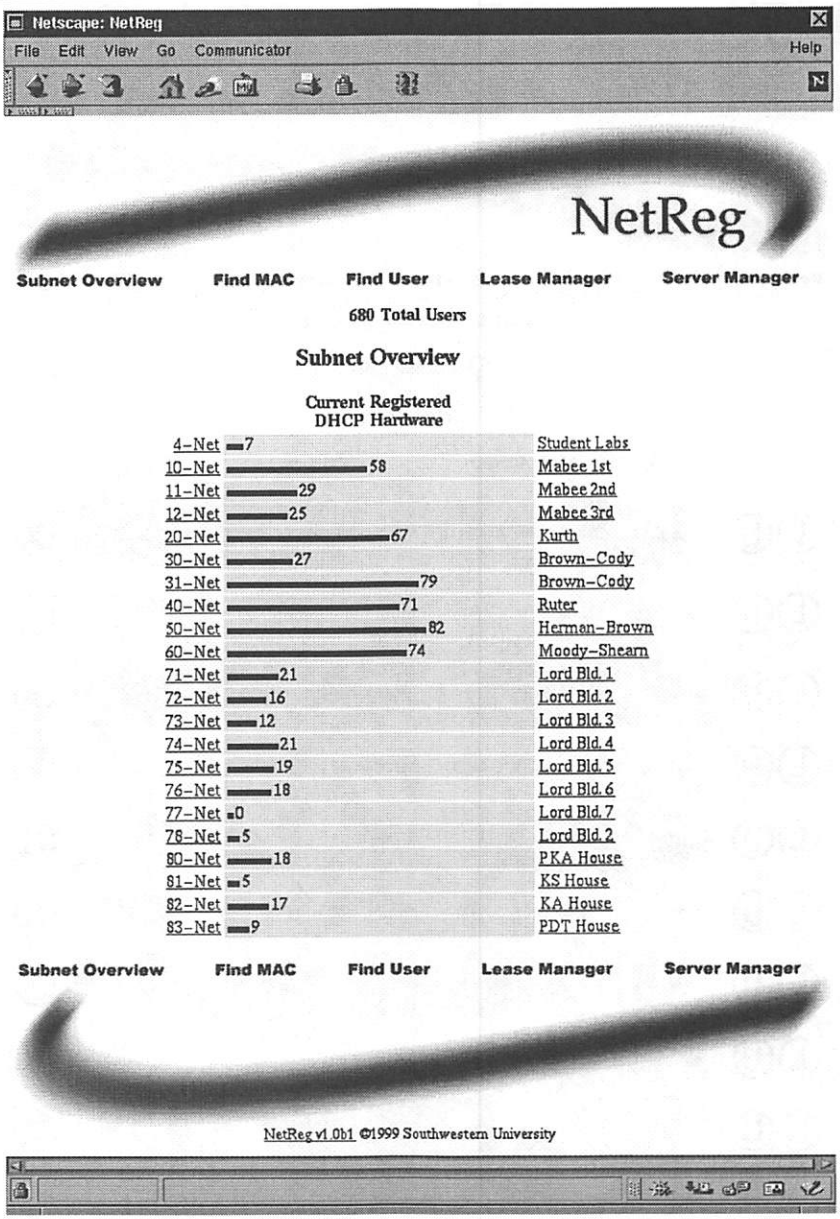


Figure 4: The Main NetReg Admin Page.

for the identity of that user ID against our internal directory server. A pleasant surprise came out of the "Find MAC" routine. Actually we can search for any string, for instance we can search for the string "PPC" or "Linux" to find useful statistics on non-Intel or Linux systems that are registered. Simple scripts can be written to get these data, but a quick-look can be done with the search function, which is very useful.

Finally, we have the ability to **delete** a registration entry via the web interface. By doing this, when the DHCP client requests a renewal of its current lease it will be denied a renewal and offered a new temporary lease from the "unknown" address pool and forced into the registration process again. If the case arises where an ethernet card changes ownership, the computer with the card will be allowed access to the network until we force another registration. This can be done when we flush the registrations each semester or if we discover the computer is roaming for an extended period from its original location[10]. Since many students relocate each semester, we plan to force them to re-register each semester[11].

These features put the power of managing the registrations into the hands of the people who need it, without having them editing configuration files by

[10]We have a simple script that provides a list of computers roaming from the network segment they registered on.

[11]We may relax this policy to just require them to re-register at the beginning of Summer and Fall since that is when the majority of students change rooms.

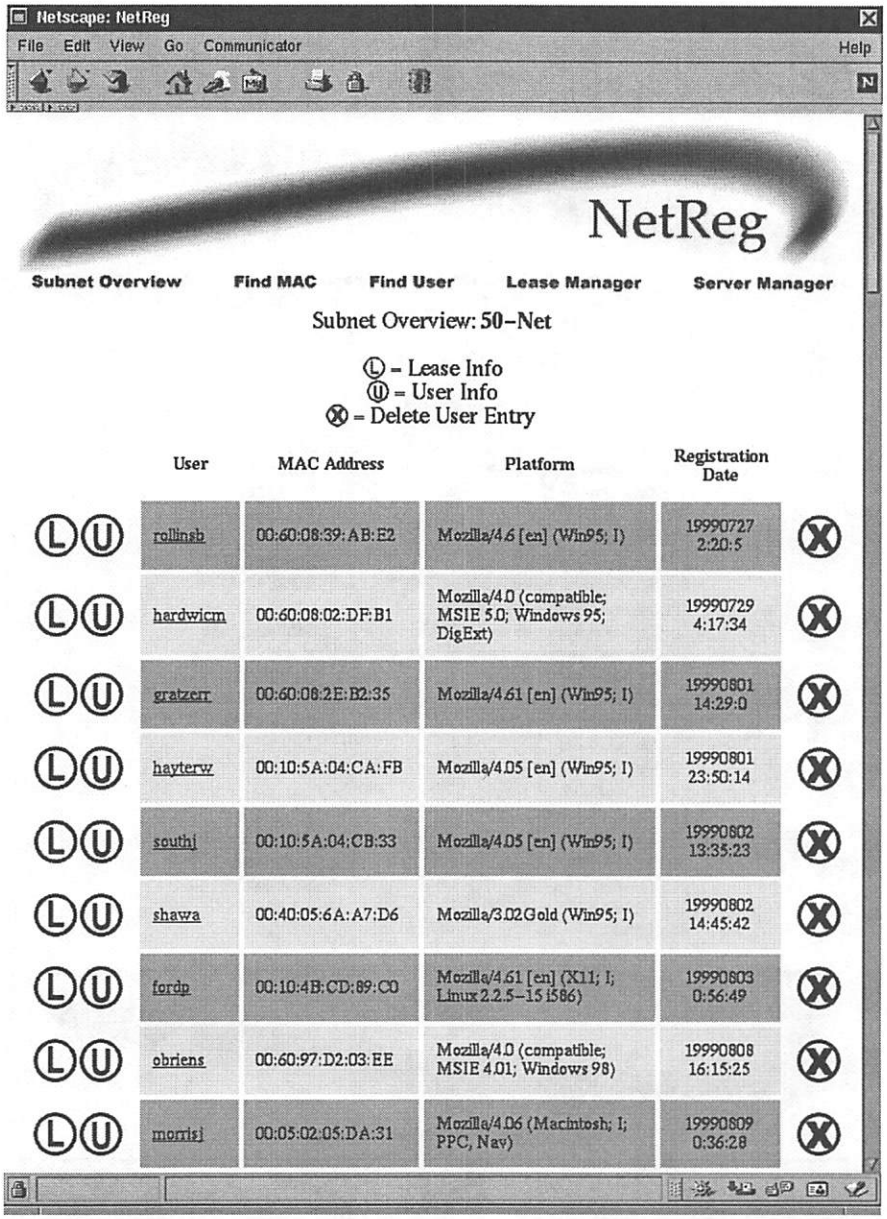


Figure 5: A View of Registrations on a Particular Subnet

hand on a system that they don't need administrative access to. Of course, due to the obvious security issues, entry to the admin site is protected by a two-layer[12] *htaccess* file [1].

Pitfalls and Lessons Learned

Even though the development of NetReg went very fast and was fairly uneventful, we did learn a few lessons which we should share. It is not a great idea to be setting up "root-level" authoritative DNS servers on a whim on your network, but there are some safety tips for doing so. We used the "Bogus Name Server" feature of BIND [2] to tell our legitimate campus DNS servers to not communicate with our NetReg DNS server.

We hoped to not have to require the client to reboot for the new lease to be obtained. In actuality, we were able to get the clients to obtain their new lease without rebooting using the network control panels on the various OS's. This did not appeal to us since the instructions for this process varied greatly based on the platform, so we opted to use the simple process of rebooting. There was another reason too. NetReg checks for new registrations with cron each minute so there is a possibility that a client who registers at one second past the minute will have to wait as much as 59 seconds until their registration entry appears (or they will obtain another temporary "unknown client" lease). At first we planned to have the actual registration CGI program just restart the DHCP server with the new registered client in the configuration file, but this created security issues we did not want to address. The problem is that the CGI program runs as the non-privileged UID of the web server, whereas the DHCP server runs as 'root'. We therefore chose to just have cron check for the new registrations. Because cron has a lower limit of one minute on how often it will run, and because boot times are getting quicker and quicker, we may have to re-address this in the future. For now the solution is that if the client is rebooted before the server reloads[13], they will be taken back to the registration page which will tell them they are already registered and should reboot their computer (again).

Another lesson we learned was about how browsers cache pages. Since the browser's default homepage gets redirected to the NetReg registration page the first time a client boots on the network, it caches the registration page for that address. Once the machine is registered and reboots, the browser reloads the cached version of registration page for the default homepage, which is not correct. Browsers should respond properly by using the "reload" button, however, our tests showed they didn't always do so. We remedied this problem by including two META tags in the header of the HTML of the registration page:

```
<META HTTP-EQUIV="Expires"
                        Content="0">
<META HTTP-EQUIV="Pragma"
                    Content="no-cache">
```

These tags seem to fix the reload problem on Netscape Communicator and MS Internet Explorer[14].

One pitfall of our current configuration is that the NetReg system runs on a single system. This single point of failure could be disastrous for extremely large sites with very short lease times and high rate of registrations. We plan to eliminate the possibility of failure by implementing multiple servers when the ISC DHCP server supports multiple server configurations. Until then, we have our previous DHCP server configured for operation and only need to load a different configuration on our router to put it back into operation taking over for the NetReg system should a fatal error occur. Also, being a low-end Pentium system running a free OS, we can easily have a replacement up in a short time period.

Security

Since much of the intention of NetReg was to improve security (mostly the management of security-related data), we should mention a few security-related issues we have and have not addressed. One of the more recent hot discussion items in the security realm is "MAC Spoofing." This is a process where a user modifies the MAC address of their ethernet card. This type of change could lead to some interesting events, such as denial of registration for another machine with the same (but valid) MAC address, or denial of service attacks by being given a duplicate IP address by the DHCP server. Also, with this technique one could exhaust all available IP addresses from the server. These such instances could be quite troublesome for a network administrator to track down. These types of attacks could exist with nearly any DHCP system however. Unfortunately, there is currently no known method[15] for the DHCP server to identify a conflict in MAC addresses (since theoretically they are supposed to be unique). It is even questionable whether this is something that servers should be able to test as opposed to the network routers and switches.

On a similar note, what happens if someone decides to not follow our rules and alters their computer's network configuration? For instance, we know it is not difficult to just enter an IP address. A relatively computer literate individual can learn your network topology and addressing scheme without too much difficulty and time. How do you deal with those

[12] We use both **Limit** and **AuthConfig**.

[13] not extremely likely since they would have to have a fast boot time and have registered very close to the start of the minute

[14] We haven't had an opportunity to test all versions of browsers on all platforms. We just picked what was "common" in our environment.

[15] Known to the authors who have kept up with the DHCP server developments fairly well.

individuals? We have a fairly crude way of handling this. We wrote a script that will find "rogue" addresses by regularly comparing the IP address leases of registered MAC addresses with MAC addresses that appear in our router's ARP table. We do this with SNMP and Perl in our script which we appropriately named *findrogue*.

Another thing a "belligerent" person may try in order to bypass NetReg would be to enter a valid DNS server (instead of our "fake-root" server). This can be deterred by using non-routable (or "internally routable" only) IP addresses for the temporary leases. Configured this way, the router will discard any packets not destined for the NetReg server.

There are undoubtedly numerous "tricks" that one can come up with to wreak havoc on our system; however, we designed it with functionality in mind more than being a highly secure system. We have developed utilities such as *findrogue* to help us find any users attempting to bypass it, but it is certainly not a system that an extremely security-conscience organization would want to implement without better studying the security issues.

Wish List

When we decided to create NetReg, we had only a couple of months of non-dedicated time to put toward developing the system. We had a set of features that had to be included, but in our opinion there is much left to be done. We hope NetReg will gain features as we, along with other users, explore the potential of this system.

The first things we'll mention are those which should come as a result of development on the servers which we utilize.

1. Support for multiple DHCP servers for redundancy and load-balancing via the "DHCP Failover Protocol" [4]. This is a feature to be included in version 3.1 of ISC's server, but has not yet been implemented.
2. Dynamic DNS integrated with DHCP. This is also expected soon with the ISC server[16].
3. Better/Smarter DHCP clients are sure to come. ISC is working on implementing authentication into their DHCP clients and server. That, along with general improvements on properly releasing and renewing leases, will help people in all areas using DHCP.

The remaining items are things which we may possibly implement or hope that others will contribute.

1. We built NetReg's authentication methods around two common servers (POP or FTP), but we would like to add modules for other more secure and appropriate methods. Obvious alternatives would be to use a Network Authentication Server (NAS), such as Tacacs+ and Radius, or Kerberos.
2. As our campus network becomes fully switched to the desktop, it could be beneficial to have the NetReg server use SNMP to make dynamic VLAN assignments to ports on switches based on the profile of an authenticated user.

[16]Check the ISC web site (*http://www.isc.org/*) for the latest status of this server.

Summary

As stated, we developed NetReg for our specific needs; however, we knew many others were looking for a solution similar to ours. It is our hope that NetReg will help those who are looking for simple DHCP management. We have done our best to keep portability in mind during development. The code was written in a very modular format with hope that others will contribute better modules and keep NetReg in an ever-developing state. A handful of other universities have put NetReg into production on their network. We hope that it works well for them, and also hope others will find it useful as that was our intent on making it available to the public.

Availability

NetReg has been released under the GNU Public License (GPL) and is available at: http://www.southwestern.edu/ITS/netreg/. Also, discussion and announcement mailing lists have been created. Information about subscribing to them is available at the above mentioned URL.

Acknowledgements

We would like to thank Bob Paver for both funding and allowing them to pursue this project. Thanks to Rich Graves for pointing out the obvious! Thanks to Pat, Rich, Sharon, and Traci for putting up with us and the phone calls! We would also like to thank Bob Horick for his listening ear and suggestions from the first ideas of this project.

Peter would like to thank his advisor (and friend) Dr. Walter M. Potter. . .thanks for always believing in me.

Todd would like to especially thank Stephanie, Nathan, and Jacob for accepting his absence (physically and mentally).

Author Information

Peter Valian <valianp@southwestern.edu> is currently an undergraduate at Southwestern University majoring in Computer Science with graduation expected May 2000. When he's not studying or sleeping, he gets to play with cool networking toys and will hopefully develop a script or two to simplify the administration of those toys. Peter's interests are network administration, network security and making Perl do cool things. You can learn more about Peter at http://www.southwestern.edu/~valianp/.

Todd K. Watson <tkw@southwestern.edu> is the "Systems and Network Administrator" for Southwestern University since 1997. At Southwestern he has the privilege of working with some really talented people and faculty. Previously he was pursuing the life of a poor Astronomer. He also worked for a short period at a really cool place called the International Institute of Theoretical and Applied Physics located at Iowa State University. His interests are in Astronomy, programming, systems security, and Unix/Linux advocacy. He has successfully avoided running NT on any system that has any importance. His almost always out-dated homepage can be viewed at: http://www.southwestern.edu/~tkw/.

Bibliography

[1] Ben Laurie & Peter Laurie. *Apache: The Definitive Guide*, Second Edition. Sebastopol, CA: O'Reilly & Associates, Inc. 1999.

[2] Paul Albitz & Cricket Liu. *DNS and BIND*, Third Edition. Sebastopol, CA: O'Reilly & Associates, Inc. 1998.

[3] Ellen Siever & David Futato. *Perl Module Reference*, Volume I. Sebastopol, CA: O'Reilly & Associates, Inc. 1997.

[4] Internet Engineering Task Force (IETF). *DHCP-Failover Protocol*, Internet Draft, http://www.ietf.org/internet-drafts/draft-ietf-dhc-failover-04.txt. expires Dec. 1999.

[5] Internet Engineering Task Force (IETF), *Dynamic Host Configuration Protocol*, Request for Comments (RFC) 2131, http://www.ietf.org/rfc/rfc2131.txt, 1997.

[6] *The Apache SSL Project*, http://www.apache-ssl.org/.

[7] *The Open SSL Project*, http://www.openssl.org/.

Dealing with Public Ethernet Jacks – Switches, Gateways, and Authentication

Robert Beck – University of Alberta

ABSTRACT

This paper describes the tools and techniques developed and deployed to address the problem of blocking unauthorized users on unprotected Ethernet jacks. Our solution is being deployed to control public labs at the University of Alberta during the summer of 1999. In this environment, we have a mix of "walk up" Ethernet connections used for laptop computers, and public Windows 95 and 98 workstations with fixed Ethernet connections. By themselves, none of these provide adequate facilities for preventing unauthorized Internet usage and enabling us to track Internet abuses originating from these networks. Prior to the deployment of our new access control system, these networks were not routed off of our campus due to these problems.

Our access control system consists of MAC-locked switches behind a gateway at which an IP filter only allows Internet access when authenticated. Now we allow the authenticated users full access to the Internet, while preventing unauthorized people from plugging in for free Internet access. This also provides a record of Internet activity by authenticated users so that abuses can be easily tracked.

We also have several transparent proxies on the gateway machine to assist us in handling particularly troublesome security and configuration issues relating to the internal lab. This allows us to selectively proxy out bound IMAP, SMTP, and HTTP requests, as well as answering IDENT requests coming in to the lab with the real user. The solution is inexpensive and easy to deploy, using off-the-shelf switches and a gateway router running a free operating system and software.

The Problem of Public Labs and Unsecure Ethernet Jacks

Public labs and their use by students in a university environment have always raised several issues for network managers and those concerned about security. The first goal of the labs is always to allow students to learn, and offer them every amount of freedom possible to do so. At the same time, there is great potential for abuse where public lab hosts can be used to attack other sites. If there are no access restrictions and no monitoring, people off the street can walk in and obtain free Internet access. University students can also usually be counted on to take advantage of a chance to cause mischief. This is especially problematic if it is unlikely that they can be held accountable for their actions.

Before the days of laptops, this problem had been addressed in our environment by allowing unrestricted net access only via hosts running multiuser operating systems such as Unix, assigning logins, and using a variety of tools to enable the behavior of the users to be monitored to some degree. This also ensured that only someone in possession of a legitimate login id and password could use the facilities. In this traditional model, system administrators maintained administrative control of the hosts, and ran an Operating System (OS) allowing them to limit and monitor the activities of the users. Access to the Internet was controlled by access to the machine. With the more common use of laptops and the requirement to run unsecure PC desktop operating systems like Windows 95/98 in public places, this solution was no longer feasible. A new solution was needed that preserved the freedom of users to access the Internet, but maintained the ability to associate specific network traffic with specific users, if necessary. This solution also had to accommodate users plugging their own computers into the network.

The Requirements for a Solution

We decided early on that a solution for us would need to have the following properties:

- It must be easy and inexpensive to deploy
- It must work with a default configuration of Windows 95 preferably with no additional software required, and as little user education as possible.
- It should work with our existing authentication methods on campus (Unix and AFS/Kerberos [3, 7, 10]) – We don't want to hand out another 50,000 login accounts.
- It should not allow unauthenticated users to use the Internet
- It should be as unrestrictive as possible to the users once they are authenticated.
- It should give us the ability to track the user id behind Internet abuse at least as well as we can

track the same for the users of our multiuser UNIX hosts.
- It should work both for locations with unsecure Ethernet jacks and for public PC labs.

Other sites have approached a similar problems by having students register MAC addresses through some mechanism before the user can receive an IP address via DHCP [11], or by the use of commercial firewall software [12, 13, 14].

The commercial firewall solution used by many locations was examined and found wanting for several reasons. The foremost reason was cost – every product we examined was prohibitively expensive for us. Secondarily, the solutions we looked at required either one of two things:
- The user had to do something (usually a telnet and authenticate) to actively "log in" to the firewall to authenticate from an IP address then had to do something else (telnet and authenticate again) to actively "log out". This was a problem for us both from a user training point of view, and from a security point of view (session spoofing prevention is difficult).
- If the above issue was addressed, it was usually by means of a custom modification to the client host's operating system. We could not support these on the scale that our campus would require.

The DHCP MAC registration techniques used by other sites were also considered. Again, we had some difficulties with implementing this at our site, Specifically:
- We now would have to register and support the MAC addresses for all our users, in addition to our existing support for 50,000 campus-wide login ID's – this was a big support issue to us.
- MAC addresses are broadcast for DHCP requests, even on a switch – they are hardly secrets – an attacker can simply configure their computer to use another user's MAC address which they have seen before on a broadcast thereby effectively becoming that user, with us being none the wiser unless the attacker and the victim are attempting to use the system at the same time.
- Our policy, for better or worse, tells our users that they are responsible for keeping their password from anyone else or suffer the potential consequences. We felt it would be unreasonable to add their Ethernet MAC address to the list of secrets that a user was responsible for protecting.
- This solution does not work for labs of PC workstations, so we would need to deploy a different solution there.

Once we decided these solutions did not fit our needs we then looked at implementing our own solution.

Our Solution

Our solution uses the following key pieces:
- An authenticating gateway router. One of these sits in front of each lab.
- A switched Ethernet network consisting of cabling and Ethernet switches.
- Our central Kerberos servers for user authentication.

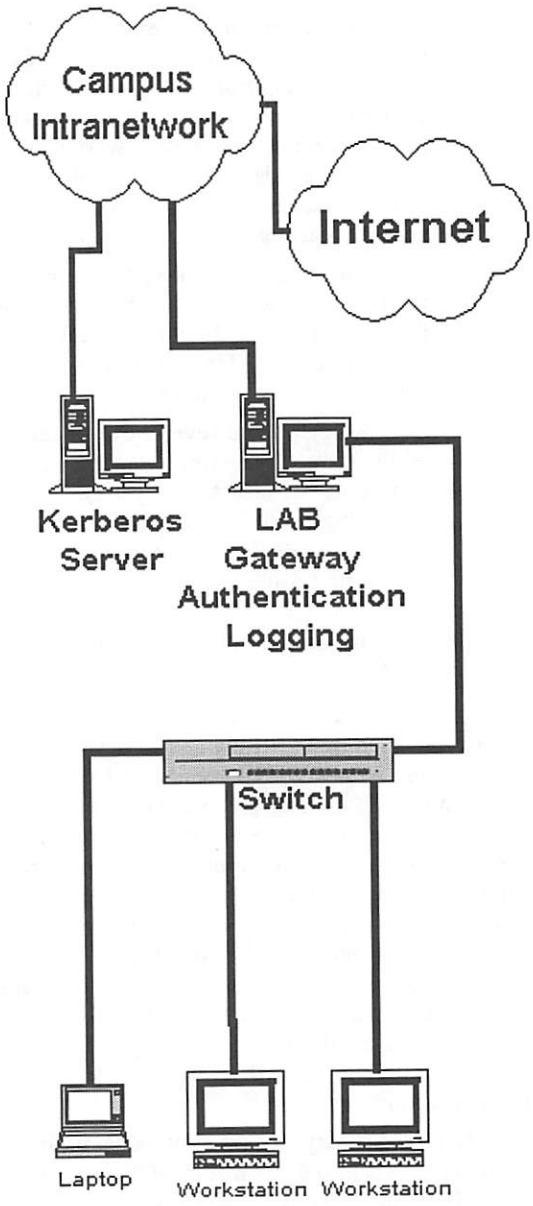


Figure 1: Authenticating gateway.

Figure 1 shows the typical setup in our environment, with the authenticating gateway located at the perimeter of the Lab network. The authenticating gateway router is configured using packet filters to by default block all traffic from the lab network. The switches in the lab allow the user to connect to the gateway router. A user gains Internet access by

opening a telnet connection to the gateway, and then authenticating with a Kerberos id and password. Successful authentication automatically creates packet filter rules that permits the user to route through to the rest of the campus and to the Internet, as long as the telnet connection remains open. When the authenticating telnet connection is closed, the filters are removed, so Internet access from the connecting address is no longer allowed. The gateway logs all the TCP connections to the Internet, as well as the login/logouts of each user.

Switches

All our public labs have been configured to use a switched Ethernet network. Properly configured, these will normally ensure that traffic will remain separated, and that a user on one port of the switch will not be able to see or gain control of traffic to or from another port of the switch. This is a requirement for our solution, as we use a standard telnet connection for the authentication of the user.[1] Without this our users would be vulnerable to session snooping and hijacking of the telnet connection used for authentication.

In order for the switches to provide the needed level of security, the switches need to have certain features and be set up correctly to use them. For the labs consisting only of PC workstations, the configuration of the switch is relatively simple. We set our switches to enable *sticky learned* port security, effectively tying each port to one Ethernet address. The port is then disabled if the Ethernet address is changed [1].

For the unsecure Ethernet Jacks used with a laptop, this doesn't work. We need to have the ability for an Ethernet address on a port to change, so the switch must be set to learn new addresses as someone plugs a new machine into a port. This creates a problem. Most switches (including ours) will by default flood unknown unicast packets to all ports. This means that if an attacker can make a legitimate user's Ethernet address unknown (usually by sending many new addresses down a port for a switch to learn) he will then see traffic for the legitimate user as it is broadcast out all ports. This can cause problems with our use of an un-encrypted telnet session. To prevent this we disable unicast flooding on the switches. [1] With unicast flooding disabled, unknown unicast packets are dropped by the switch, changing this sort of attack from a potential compromise of user passwords and authentication to a merely annoying denial of service.

[1]We could use a secure connection, such as ssh, but PC OS's don't come with such software.

An Authenticating Gateway

For the gateway we had to implement software to authenticate users and manage the packet filters. We chose to do this on OpenBSD [15] as it is free, has all the relevant pieces we need in the OS, an emphasis on source code audited for security, and free source code that we can modify to suit our needs. We configure the ipf packet filtering facilities in OpenBSD to, by default, block all traffic originating from the lab side of the network. We then wrote a small program and modifications to /usr/bin/login so that the users could telnet to the gateway and authenticate to get Internet access. To facilitate the client setup, we use an interface alias on the internal interface of all our gateways to answer to a private [6] IP address – 10.0.0.1. This lets us configure any client to telnet to the same place for authentication, and they will always connect to the correct gateway machine for their current location on our network.

By default, our packet filter rules on the gateway are set up similar to the following in /etc/ipf.rules; see Listing 1. When the user authenticates, a program authipf is run which does the following:

1. Log the fact that the user has connected and authenticated.
2. Add filter rules to allow all traffic from the connecting IP address, logging all TCP connections.
3. Wait for the connection to go away
4. Removed the filter rules from above
5. Log the fact that the user has disconnected, and the duration of the session.

```
## allow the inside to talk to ME, to telnet for authentication
pass in quick on fxp0 from any to 129.128.38.65/32

# We use an interface alias on the internal address, so all clients
# can be configured to telnet to one address and they aren't
# different in each lab.
pass in quick on fxp0 from any to 10.0.0.1/32

# log all inbound TCP connection requests
pass in log quick on ep0 proto tcp from any to 129.128.38.64/26 flags S/SA

# otherwise deny anything in on internal interface from the internal net
# that does not get allowed by subsequent (added by authipf) rules.
block in on fxp0 from any to any
```

Listing 1: Packet filtering rules.

authipf adds two filter rules similar to:

```
pass in log quick proto tcp
    from 129.128.38.100/32
    to any flags S/SA
pass in quick
    from 129.128.38.100/32
    to any
```

authipf removes the same rules when it exits. These rules pass all IP traffic, logging the TCP connections.

For authentication, our campus already makes use of Kerberos [3, 7]. We have an existing user base of approximately 50,000 accounts, one for every student and staff member. We did not, however, want to manage accounts on the lab gateways, since users there did not require login access to the machine, only the ability to authenticate. We modified OpenBSD's login program slightly for our purposes. The changes were very simple:

- If the user exists locally, proceed with the normal login procedures.
- Otherwise, if the user does not exist locally, try the username/password in Kerberos. If this succeeds, run authipf for the user/IP address. Otherwise, fail.

With these pieces in place, we have everything we need to control and log the out bound Internet access from the lab, ensuring only authorized users are able to access the Internet. Legitimate users simply open a telnet connection to the gateway and authenticate. As long as the user leaves open the telnet connection to the gateway, the gateway allows their traffic to pass. As soon as the user is disconnected from the gateway, traffic is no longer allowed to pass. The user needs nothing special beyond a telnet client on their machine. Since unauthenticated traffic is not allowed out, any attempts by the users to send out packets with a bogus source address are blocked by the lab gateway. This means that we do not need to worry about this sort of activity putting put a load on our permiter routers.

A user authenticating, using the net, and then exiting might look something like Listing 2.

In this case, we see a user (beck) authenticating from a lab host (129.128.38.100), at 10:42 AM, and then disconnecting 67 seconds later, with some telnet and web activity in between time. Note that by default, we do not log non-TCP traffic (UDP, ICMP) explicitly. In practice this has not proven to be a limitation, as most abuses (i.e., ping floods, udp mischief) are easily tracked by the times they occur.

Problems and Solutions

One problem we encounter with Ethernet jacks in a public place is that of users unplugging their laptop from the net and walking away without closing the session. In this case, we need the gateway to know that the user has gone away, and to remove filter rules allowing their machine access to the Internet. We tuned the TCP KEEPALIVE values on the gateway machine to ensure that sessions would be expired promptly should the user unplug a laptop and walk away. We decided that for our purposes it was adequate for connections of this type to go away in under a minute.

Another possible problem is a user injecting bogus ARP replies into the network to take over an IP address. This is a problem, since our authentication of a user is based on the IP address they are using. OpenBSD itself watches and notices in the syslog when ARP values change. While this is a regular occurrence in a plug-in environment, it can also be used as an attack. To handle this, we can use Swatch [2] to watch the logs for the ARP entry being overwritten for a particular IP address, and then ensure that any running authipf process for a particular IP address is killed, so the IP address is not allowed to pass to the Internet until it authenticates again. While this creates a possible denial-of-service attack within the lab, it does not allow a user to inject bogus ARP replies into the network and take over an authenticated IP address to gain Internet access. As soon as the gateway detects the change in the ARP table entry, any old authentication info and filters for that IP address are removed.

Another risk to keep in mind is the fact that in this environment we use switches configured with

```
Aug  1 10:42:03 law104-gw authipf[27571]: Allowing 129.128.38.100/32,
                                                        user beck
Aug  1 10:42:14 law104-gw ipmon[17403]: 10:42:13.826803 fxp0 @0:27 p
129.128.38.100,1049 -> 129.128.125.13,23 PR tcp len 20 48 -S
Aug  1 10:42:14 law104-gw ipmon[17403]: 10:42:13.826803 fxp0 @0:27 p
129.128.38.100,1049 -> 129.128.125.13,23 PR tcp len 20 48 -S
Aug  1 10:42:02 law104-gw ipmon[17403]: 10:17:51.737302 fxp0 @0:25 p
129.128.38.100,1064 -> 216.33.151.7,80 PR tcp len 20 48 -S
Aug  1 10:43:02 law104-gw ipmon[17403]: 10:30:11.730452 fxp0 @0:25 p
129.128.38.100,1062 -> 216.33.151.7,80 PR tcp len 20 48 -S
Aug  1 10:43:10 law104-gw authipf[27571]: Removed 129.128.38.100/32,
                                    user beck - duration 67 seconds
```

Listing 2: Authenticating, using the net, and exiting.

some unusual options. The security of this system depends on the switch being configured to either have sticky learned port security turned on, or unicast flooding disabled. This could be easily "overlooked" by people performing maintenance on the switch. Our campus uses SNMP monitoring systems as well as regular review by paranoid individuals to ensure that this risk is minimized to an acceptable level for us. One must also ensure that the switch is configured not to accept any sort of management connection from the internal network, lest someone attempt to bypass security by breaking in to the switch and changing the switch configuration.

Further Support for the Lab Environment

With one gateway machine and switches, we now have a "one-box" solution to our public laptop needs. We simply run dhcpd on our OpenBSD gateway to provide DHCP [5] service to laptop clients, in addition to acting as the authenticating gateway.

In addition to merely authenticating the users, and allow/disallowing connections based on packet filtering techniques, with a gateway machine in place we also have the ability to intercept and transparently proxy certain requests. We do this in several situations.

The gateway machine proxies inbound IDENT [8] requests, and answers them on behalf of the internal clients. We run IDENT servers on most of our centrally administered hosts so other on-campus hosts may query the IDENT server to learn the identity of a remote user. On our gateways, we capture IDENT requests for the client addresses and we then answer them with the username used to authenticate out, enabling remote system administrators to query for and obtain user names using IDENT when they receive connections from our labs.

For some of our locations, we gain some additional security and ease of client configuration by storing the user and password information on the gateway and using this in transparent proxies. On the gateway we are able to catch and proxy IMAP [4] protocol connections to our central IMAP server. We have a simple proxy which watches for the IMAP LOGIN command, and replaces the arguments to the command with the user's real login ID and password, as used to authenticate from the connecting address. Connections to other IMAP servers are not captured, and so pass through unchanged. We also intercept out-bound SMTP [9] access to our central SMTP server, replacing the sending address with the user's real e-mail address (derived from the login they used to authenticate). As with the IMAP case, connections to other SMTP servers are not captured and will pass unchanged. While the ability to do this has some interesting security implications, we do it mainly to allow easy set up of an e-mail client in the PC labs, so that the configurations of the client don't need to change from user to user.

Problems This Doesn't Solve

Many people have been, and continue to be, concerned that this does not address the issue of "What if the user gets up and walks away." The filter rules are not tied to any sort of activity monitoring to detect an active or idle connection and timeout, although authipf has an optional overall timeout value. We chose not to implement an activity based timeout for several reasons:

- We don't think users will be likely to leave their laptops and walk away, so it's only a problem when using labs of PC's.
- It doesn't solve the problem. If the user walks away someone can simply pick up where they left off before the timeout value.
- It's annoying to the users to constantly have to re-authenticate if they are periodically accessing the Internet for information, while doing their own work which does not involve Internet access.
- We already are well versed in dealing with the problem of users leaving themselves logged in on the regular timesharing systems – we deal with this effectively already as a "People Problem" that doesn't require a technical solution.

This does not address several denial-of-service issues or problems of intra-lab abuse (from one workstation to another). These are addressed by more traditional (technical and non-technical) means in our labs.

It is important to remember that this is a method for providing authenticated Internet access. It does little (or nothing) for the security of the individual client stations. (If the bad guy is already on the user's laptop or PC, this doesn't help much).

Deployment and Use

Our initial test deployments of this system were deployed in front of networks in our student residences, and in front of one large scale laptop Ethernet jack deployment in September 1998. It has served us very well with very few problems. As of Summer 1999 we are deploying this system in front of all of our public PC labs and public Etherenet jacks. Important to know is we typically do not actively monitor what people are doing in the labs looking for problems. If we don't receive complaints, then we don't really care what students are doing. On the other hand, when we do receive a complaint about something originating from one of these locations, finding the offending user has been a very simple process of looking in the logs for the evidence at the right times. If we had wanted to restrict wholesale what users were able to do, we would have taken a different approach, likely something more akin to a traditional firewall.

Currently we are the process of deploying this system in front of 35 lab locations (20-30 workstations each) throughout our campus, as well as a number of other laptop plug locations. The deployed gateways

have disk adequate to store log records for over six months, which is as much as we feel we will need for our purposes.

Conclusions

We have so far been very happy with the level of control this system has given us when deployed in front of our unsecure public places. It is entirely based on free software, works well in our fully switched environments. It integrates well with our Kerberos authentication (and could be easily made to work with others). It allows the users full network access, and does not require any custom software on the client hosts at all. User training has been relatively painless, consisting of a poster at the front of the lab, as well as an icon added to the standard desktops of the workstations. The system completely prevents unauthenticated users from using public Ethernet jacks to gain Internet access. The logging produced as a bonus is thorough and usable to track abuse by authenticated users very effectively when a complaint is received. With the addition of the IDENT proxy, many complaints have already been resolved without even resorting to the logs ("Yes, you can believe the IDENT – thanks"). If you are looking for a solution to dealing with public Ethernet jacks and Internet access, this is a great solution that scales very well, at very little cost.

Availability

Our code is available under BSD-style license terms, and can be obtained from ftp://sunsite.ualberta.ca/pub/Local/People/beck/authipf/ Our gateways were all built using OpenBSD, Much of the code used to construct this system was taken from OpenBSD and modified to suit our needs. See http://www.openbsd.org for OpenBSD.

Author Information

Bob Beck has a Masters degree in Computing Science from the University of Alberta. He has worked in a variety of systems administration and programming positions at the University of Alberta since 1990. He also works as a consultant, instructor, and programmer with Obtuse Systems Corporation. He is currently the Secure Systems Specialist for the University of Alberta, as well as working on several free software projects. You can reach him by postal mail at Computing and Network Services; 352 General Services Building, U of A Campus, Edmonton, Alberta, Canada, T6G 2H1. You can reach him by e-mail to beck@bofh.ucs.ualberta.ca, or beck@obtuse.com.

References

[1] *Catalyst 1900 Series Installation and Configuration Guide* Part Number 78-4362-01, 1997, Cisco Systems Inc.

[2] Stephen E. Hansen, E. Todd Atkins, *Automated System Monitoring and Notification with Swatch*, USENIX Association's Proceedings of the Seventh Systems Administration (LISA VII) Conference (1993) p. 145-155.

[3] J. Steiner, C. Neuman, and J. Schiller, *Kerberos: An Authentication Service for Open Network Systems*, Usenix Association's Conference Proceedings, Dallas, Texas, February, 1988, p. 191-202.

[4] M. Crispin, *RFC2060: Internet Message Access Protocol – Version 4rev1*, December 1996.

[5] R. Droms, *RFC2131: Dynamic Host Configuration Protocol*, March 1997.

[6] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear *Address Allocation for Private Internets*, February 1996.

[7] J. Kohl, and C. Neuman *The Kerberos Network Authentication Service (V5)*, September 1993.

[8] M. St. Johns, *RFC1431: Identification Protocol*, February 1993.

[9] J. Postel, *RFC821: Simple Mail Transfer Protocol*, August 1982.

[10] AFS filesystem information from Transarc Corporation, http://www.transarc.com/Product/EFS/AFS/index.html.

[11] ResNet at University of California, Davis, http://resnet.ucdavis.edu/.

[12] ResNet at University of Auburn, http://webserv.duc.auburn.edu/hotline/Contents/Resnet/.

[13] ResNet at Buffalo, http://wings.buffalo.edu/computing/documentation/win/CampusAccess/firewall.html.

[14] ResNet at Wright State, http://www.cats.wright.edu/catsweb/residence/.

[15] The OpenBSD project, http://www.openbsd.org/.

NetMapper: Hostname Resolution Based on Client Network Location

Josh Goldenhar – Cisco Systems, Inc.

ABSTRACT

Large buildings, sprawling campuses and multiple remote sites have led to an explosion in the number of IP networks for corporate computing. Workgroups are spread over multiple networks. Servers are configured with multiple network interfaces in an attempt to optimize access. When geographic or capacity issues arise, separate servers which replicate desired functionality are placed in network proximity to their clients. Some of these servers provide tool trees which have operating system (OS) and architecture specific binaries. The optimal server or server interface for a given client may or may *not* have a presence on the client's network. In such an environment, how can an administrator guarantee a network client is utilizing the desired network interface or server?

NetMapper provides a framework for resolving hostnames (real or virtual) based on the client host's location within a network hierarchy. For servers with multiple network interfaces, NetMapper chooses the best interface. For multiple servers providing replicated services via a virtual hostname, NetMapper chooses the best server. For file servers providing OS and architecture specific filesystems, NetMapper chooses the best server taking into account client OS, architecture and network attributes. In all cases, 'best' is defined by the NetMapper administrator. As a side benefit, NetMapper allows systems and network administrators to view their network hierarchy at-a-glance.

The core of NetMapper functionality is a Perl module (NetMapper.pm) and a configuration daemon (nmconfd) which serves the configuration information to NetMapper clients. NetMapper has been implemented in the Cisco engineering environment by developing a small program called localmapper which runs on all UNIX clients and generates entries in the client's local /etc/hosts file. localmapper optionally generates a small NFS automounter map for OS and architecture specific remote partitions. This fairly small collection of tools allows systems administrators to choose which servers UNIX network clients use based on geography, workgroup or any arbitrary rationale that can be defined via groups of networks.

Introduction

Imagine working for a company that has thousands of UNIX computers distributed over hundreds of networks at various sites worldwide. All workgroups demand the ability to NFS mount each other's servers. Automounter maps contain thousands of keys. Add to this picture a mixture of license, compute, revision control, NTP and various other servers which are scattered throughout these networks. Lastly, imagine a user base that demands their access to these services be based on workgroup, network and geographic topologies. In this age of explosive growth for technology companies, many systems and network administrators don't have to imagine such a situation, it is reality.

As a network environment grows, it is common to replicate services by adding additional servers. Multiple network interfaces are added to servers to increase network bandwidth or provide streamlined access for a set of clients. As soon as an additional server or interface is brought online, a question arises: How does the administrator ensure network clients are utilizing the preferred distinct server or interface?

There are several existing products and methods that aid in the optimal resolution of server hostnames. Usually, these solutions make their choices based on a measurable quantifier such as number of simultaneous connections, network load or latency. These methods are acceptable when there is no differentiation between servers (such as replicated WWW servers).

In today's corporate compute environment there are often reasons to differentiate servers based on abstract policies. What if only certain file servers contain the OS and architecture specific binaries a client needs? Geographic, political, departmental or other arbitrary conditions sometimes necessitate a group of clients connect to a certain server based on those conditions. If these conditions can be split along network divisions, NetMapper provides a solution where server hostnames can be resolved in a somewhat arbitrary nature based on the desires of the administrator.

Existing Solutions

Hostnames that resolve to multiple IP addresses usually fall into one of two categories: multiple interface hostnames or virtual hostnames. Within the scope of this paper, these terms are defined as follows:

Multiple interface hostnames are defined as real hosts which have multiple active network interfaces. The hostname itself resolves (using the name service a site chooses) to one or more of the IP addresses of its interfaces. Additionally, each interface should have a distinct interface specific hostname. In Figure 1, resolving the hostname 'masses' would return both interface IP addresses. The hostname 'masses-115' would resolve to 123.45.69.115. The hostname 'masses-156' would resolve to 123.45.68.156. Thus, the administrator has a straightforward way of specifying an individual interface without having to use dotted notation.

Virtual hostnames are those that are intended to resolve to the IP address or hostname of one of a set of real hostnames. In Figure 1, two hosts ('waiter' and 'bus-boy') serve a replicated file system called /coffee. A virtual hostname 'folgers' could resolve to the name or IP address of either 'waiter' or 'coffee'. There is no computer actually named 'folgers'.

There are several existing methods and products to aid in the optimal resolution of hostnames that resolve to multiple addresses. These methods do not make a distinction between multiple interface hostnames and virtual hostnames. Some of these existing methods are described below as well as the reasons that NetMapper was chosen over them.

Domain Name System (DNS) Solutions

DNS [1] and BIND [2] solutions are most often used to resolve a multiple address hostname. While these work well for general conditions, they are less than optimal for returning addresses based on the client's network address.

It is possible to tailor BIND and DNS to return addresses based on client network location. Often, this requires many sub-domains or relies on a questionable BIND feature which returns an appropriate response based on the requester's IP address [3,4]. The latter feature has come and gone from BIND in different versions and is not viewed as reliable over time. In general the network address of the client requesting name resolution is not considered by the name server and thus answers are not tailored to the client [5]. While DNS servers have knowledge of the requesting host's IP address, they do not necessarily have any knowledge of other network interfaces on that client. Thus, even a modified DNS server would only be able to consider the source address of the request – even if the client has another interface that might provide a better path to the server hostname in question.

There are occasions when a server interface address should not be advertised by DNS for general use. Figure 1 shows two hosts, 'waiter' and 'patron', which are linked by a point-to-point connection. 'Waiter' serves /coffee to 'patron'. Optimally, when 'patron' does a name lookup on 'waiter', the IP address of 'waiter-pp' would be returned. DNS should not return the point-to-point address of 'waiter' to any other host. Therefore, the point-to-point address can't be included in the DNS configuration as an address ('A' record) for 'waiter'.

Lastly, DNS has no knowledge of the requesting client's OS or architecture. Therefore, it can't take these elements into consideration when returning an address for a hostname.

Automounter Variables Solution

If you want to take client OS and architecture needs into consideration for NFS [6] mounting, automounter [7] variables are a suggested solution to this problem. Automounter variables can be initialized at daemon run-time to change mount paths and server names. This method suffers from lack of compatibility across platforms and the fact that it only addresses the issue of hostnames for automounted NFS services.

Commercial Solutions

One solution that is applicable for practically any IP based protocol is Cisco's LocalDirector [8,9]. This is a great product for pure load-balancing. It works

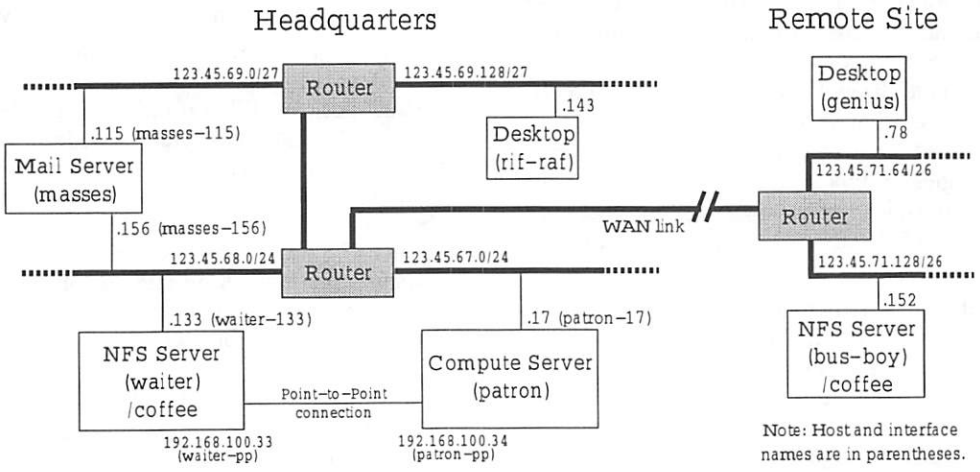


Figure 1: Example Hosts and Network Topology.

especially well to simply have network traffic automatically divided between multiple servers which all provide equal functionality. All traffic from the clients to the server(s) passes through the LocalDirector unit. This works well when the servers in the "pool" are in equal network proximity to the LocalDirector unit and the client network location is not an issue. LocalDirector can't take client OS and architecture needs into its consideration of which server to direct traffic to. While we use LocalDirector for certain applications, it can't cope with all the conditions we require.

Cisco's own DistributedDirector [10,11] overcomes LocalDirector's shortcomings in regard to network topology issues. DistributedDirector works as a DNS server. It uses knowledge of routing tables and network latency to resolve hostnames on a per-client basis. This solution is a great product for load balancing and directing traffic to hosts which provide distributed services. Because it operates as a DNS server though, it could not be used for the same reasons explained in the 'Domain Name System (DNS) Solutions' section above.

'Home-Grown' Solutions

Our previous solutions for optimally mounting NFS file systems involved a convoluted process using rdist [12], a Perl [13] script and m4 [14] to customize the auto.indirect automounter map on every client. This method worked for our largest automounter map but did nothing for other maps, processes or general name lookups. As the number of networks and hosts grew large it became obvious that this method was becoming too difficult to maintain.

After determining that no one existing solution could cope with all of our requirements, it was decided to build upon the beginnings of our own home-grown solution. NetMapper built on the groundwork of earlier attempts to solve this host/interface selection problem. The NetMapper methodology can be used as part of any number of solutions to host/interface resolution and location problems. The remainder of this paper will focus on a suite of scripts which address the problem of how a UNIX network client chooses the appropriate hostname or host specific interface when there are multiple possibilities.

Design Phase – Goals

In the NetMapper design phase there were several goals for base level functionality. The goals, and the reasoning behind them are explained in more detail below.

- A mechanism that ensures UNIX clients connect to the desired interface of a multiple interface server, or to a specific server for distributed and/or replicated services.
- Server hostnames that serve architecture or OS dependent partitions need to be selected by partition availability as well as client network location.
- Distribute as little as possible in the way of programs, source and configuration files to the clients.
- An easy to use administrative interface with basic error checking. The interface should aid in understanding the concepts behind the solution.
- The mechanism for choosing a hostname had to be well defined.

The first goal was one of the easiest to formulate, yet hardest to meet. We had to use a method to resolve hostnames that was fairly simple, yet worked on a wide variety of UNIX platforms. We did not want to implement a solution that involved replacing vendor libraries or significantly altering the operating system.

The second goal arose from the fact that in our environment we have NFS servers that provide architecture and OS specific binary tool trees. These servers are not guaranteed to contain all architecture and OS variants for these trees. When resolving a server hostname that serves such trees, it is crucial that the solution consider the client's OS and architecture. Furthermore, the solution must choose a server that has the appropriate file system available.

The third goal (distribute as little as possible) came into being from experience with our previous solutions. We were distributing automounter files, m4 definition files and Perl scripts to every UNIX client. This process took too long and was often not very reliable.

The fourth goal (ease of administration) was also based on previous experience. Our previous solutions involved multiple text configuration files which were sensitive to syntax and prone to error. We needed an interface to provide basic error checking so mistakes such as invalid hostnames did not make their way into the configuration. This interface needed to simplify the concepts behind NetMapper so that new systems administrators needed much less training than was required with previous solutions.

The final piece of the puzzle was the mechanism by which hostname and interface choices were to be made. It was decided that all decisions would be based on knowledge of the overall network hierarchy. Our sites, buildings and workgroups were all split among distinct networks or groups of networks. This would make it easy to resolve hostnames by many different conditions.

Design Phase – Decisions

After clearly defining our goals, we moved to the next phase which was deciding the best methods to achieve them.

To achieve our first goal, it was decided the easiest way to optimize hostname resolution for all services across multiple operating systems was to modify the client's local /etc/hosts file. Modern UNIX

systems can be configured to use one or more methods of hostname resolution: /etc/hosts is an option to all, and is local to the client. Older UNIX systems that are not configurable default to using /etc/hosts. As long as /etc/hosts is consulted first, any program that performs a gethostbyname call will 'use' the NetMapper preferred address.

The second goal would be achieved by creating a naming convention for file systems that contain OS and architecture specific binaries. When defining a server hostname that provides such filesystems, NetMapper configuration would store which OS and architecture specific filesystems are available on that server. The client portion of NetMapper could then take this information into consideration when choosing hostnames that are defined as multiple OS and architecture file servers. This could be used to provide both hostname and path resolution for NFS mount commands.

The third goal (distribute as little as possible) would be met by making NetMapper configuration information a client/server process. The NetMapper Perl module would get its configuration information from a configuration server. If parts of NetMapper have to be installed on the client's local filesystem, they should be capable of updating themselves when necessary.

The fourth goal (ease of administration and error checking) was to be accomplished by providing a web-based administration interface. NetMapper configuration information would be manipulated via a browser to facilitate centralized administration and avoid syntax errors in the main configuration file. (NetMapper configuration would be stored in a Perl interpreted file; syntax errors could be catastrophic.) The choice of a web-based interface for configuration would allow the NetMapper network hierarchy information to be displayed in a way which would aid understanding. CGI forms and Javascript would also allow control over what information could be changed, how it would change and would assure that dependencies are honored.

The mechanism by which hostname resolution decisions would be made was to be based on a knowledge of the network hierarchy. All networks that contained potential NetMapper clients would need to be entered into the NetMapper configuration using Classless Inter-Domain Routing (CIDR) [15-18] format. Networks could be grouped together under a logical name. Logical groups would contain networks and other network groups. Each NetMapper hostname would be defined by mappings that specify which real hostnames, or interface specific hostnames should be used based on a network client's location within the network hierarchy. Architecture and OS specific virtual host definitions would contain a record of server candidates and which platforms those candidates are capable of serving. Together, these pieces would allow NetMapper to find a client's place in the network hierarchy and determine an appropriate server hostname for that network client.

Implementation

In order to achieve the goals and specifications of the design phase, NetMapper was split into parts and built on the existing infrastructure. Once all the functionality goals were finalized, the actual implementation took shape rather quickly. The design was implemented with a complete suite of Perl scripts. Details of the implementation are described below.

Enter LocalMapper

The manipulation of local /etc/hosts files is performed by localmapper. This method relies on the fact that /etc/hosts is searched from top to bottom and returns the first match it finds. Therefore, this script has a prerequisite which can have no exceptions – clients must have a static, truncated hosts file. That is, it must contain only 'localhost', its own hostname(s) and a bare minimum of other hosts.

The localmapper script appends a delimiter and all defined NetMapper hostnames to the hosts file. Thus, all localmapper modified hosts files will contain the same hostnames but possibly different IP addresses after the delimiter. If the localmapper delimiter is found in an existing hosts file, all entries after the delimiter are removed and replaced with the current NetMapper selections. This allows for overrides and other customized information to remain in the client's hosts file (above the delimiter).

The localmapper script is careful to avoid corrupting or truncating the /etc/hosts file. It uses a temporary file to construct the new hosts file. If there are any fatal errors the script aborts, leaving the existing /etc/hosts intact.

Sample /etc/hosts files based on the hosts in Figure 1 are included in Listing 1. The hostname 'folgers' is virtual – there is no actual host named 'folgers'. The hostname 'folgers' is defined in NetMapper (see Figure 2) to be either 'waiter' or 'bus-boy'. The desktop 'genius' at the remote site will be served /coffee by 'bus-boy'. Any host at headquarters will be served /coffee by 'waiter'. It should also be noted that the host 'patron' has the point-to-point IP address for 'waiter' in its /etc/hosts file, while all other hosts use the 'public' address for 'waiter'.

The localmapper script is the only program that needs to run on the client. If the localmapper script is installed on a client's local file system and is out of date, it can update itself when necessary.

OS and Architecture Specific Server Names

When resolving a virtual hostname which serves multiple architecture and OS partitions, it does no good to use the 'best' server for a network client if that server does not have the specific file system that the client needs to access.

In our compute environment we support various OS and architecture specific versions of /usr/local. There are multiple NFS servers throughout our networks which provide these OS and architecture specific file systems. On a UNIX client, /usr/local is actually a link to /auto/usrlocal. The directory /auto is an automount point defined by the auto.indirect automounter map. Thus, 'usrlocal' is an automounter key. The idea behind all this is that UNIX clients will dynamically mount /usr/local from their departmental or network closest server via a common pathname: /usr/local.

The virtual server name for the example above is simply 'usrlocalhost'. This virtual hostname must be resolved to one of several real server hostnames that serve copies of /usr/local. The problem is that not all variants of the /usr/local tree are available on every server. The NetMapper library takes this into consideration when asked to resolve such a hostname. When localmapper in invoked with the '-a' option, it produces a small indirect automounter map written to /etc/auto.netmapper. This file is overwritten if it already exists. It contains the keys, hosts and mount paths for these special file systems and servers. This allows all hosts to access these shared trees via a common name regardless of client OS or architecture.

Using the hosts in Figure 1, the /etc/auto.netmapper file for the host 'patron' is shown in Listing 2. There are various ways to make use of this map. It can be used as a separate map for a unique automount point. It could be included from another automounter map with a '+' entry. Modern UNIX systems could be configured to use this map before or after a NIS or NIS+ map.

```
# /etc/hosts file for desktop system rif-raf (Headquarters - Figure 1)
127.0.0.1        localhost
123.45.69.143    rif-raf
###### LocalMapper Auto-Generated Entries Below ######
###### Changes made below this line will be lost #####
123.45.68.133    folgers waiter
123.45.67.17     patron
123.45.68.133    waiter
123.45.69.115    masses masses-115

# /etc/hosts file for server system patron (Headquarters - Figure 1)
127.0.0.1        localhost
123.45.67.17     patron
192.168.100.34   patron-pp
###### LocalMapper Auto-Generated Entries Below ######
###### Changes made below this line will be lost #####
192.168.100.33   folgers waiter-pp
123.45.67.17     patron
192.168.100.33   waiter waiter-pp
123.45.68.156    masses masses-156

# /etc/hosts file for desktop system genius (Remote Site - Figure 1)
127.0.0.1        localhost
123.45.71.78     genius
###### LocalMapper Auto-Generated Entries Below ######
###### Changes made below this line will be lost #####
123.45.71.152    folgers bus-boy
123.45.67.17     patron
123.45.68.133    waiter
123.45.68.156    masses masses-156
```

Listing 1: Sample /etc/hosts files for three hosts in Figure 1.

```
# LocalMapper Auto-Generated automount table - changes will be lost
# key       mount options       hostname:mountpath
usrlocal    -ro,soft,noquota    waiter-pp:/export/usrlocal.sparc-sunos5
sw          -ro,soft,noquota    waiter-pp:/export/sw.sparc-sunos5
```

Listing 2: Sample auto.netmapper automount map for SunOS 5 host 'patron'.

Configuration Server

The NetMapper configuration server (nmconfd) aids in the effort to distribute as little as possible to network clients. The nmconfd script serves the Perl interpreted configuration file (netmapper.conf) to network clients at runtime. The NetMapper Perl module (NetMapper.pm) attempts to communicate with nmconfd on a hostname 'nmconfhost', port 13000.

For scalability, it is possible to have multiple NetMapper configuration servers by setting up 'nmconfhost' as a NetMapper hostname itself! The hostname 'nmconfhost' should have an entry in DNS that is the default address of the configuration server. In this manner, when localmapper runs for the first time (or has failed) a lookup of 'nmconfhost' will not result in a 'host unknown' error. (This is also the suggested way to add a default for all NetMapper hostnames.) The first time localmapper connects to nmconfd, it will use the default (DNS record) address. Once localmapper runs, later invocations will connect to the desired configuration server.

The nmconfd script can also provide the latest version of localmapper to enable self updates. The version information for localmapper is kept in netmapper.conf along with the network hierarchy, NetMapper hostnames, known architectures and OS's, interfaces to be ignored and a serial number.

Administrative Interface and Configuration Files

The information contained in netmapper.conf is stored in various Perl structures such as anonymous references to hashes and arrays. This information can be tricky to edit and is prone to syntax errors. The web administration interface was developed to greatly reduce this risk of corruption as well as presenting a more intuitive view of the network hierarchy and members (see Figure 3). The main CGI configuration script, netmapperconfig.pl, makes use of the CGI.pm, Net::DNS and Data::Dumper Perl modules for interface, name resolution and configuration file output. It was necessary to use Net::DNS for hostname resolution in case the host that was running netmapperconfig.pl was also a NetMapper client. (Calls to gethostbyname would use the /etc/hosts file for hostname resolution. If localmapper had previously modified the /etc/hosts file, the lookup of a hostname that it was trying to resolve would return whatever was

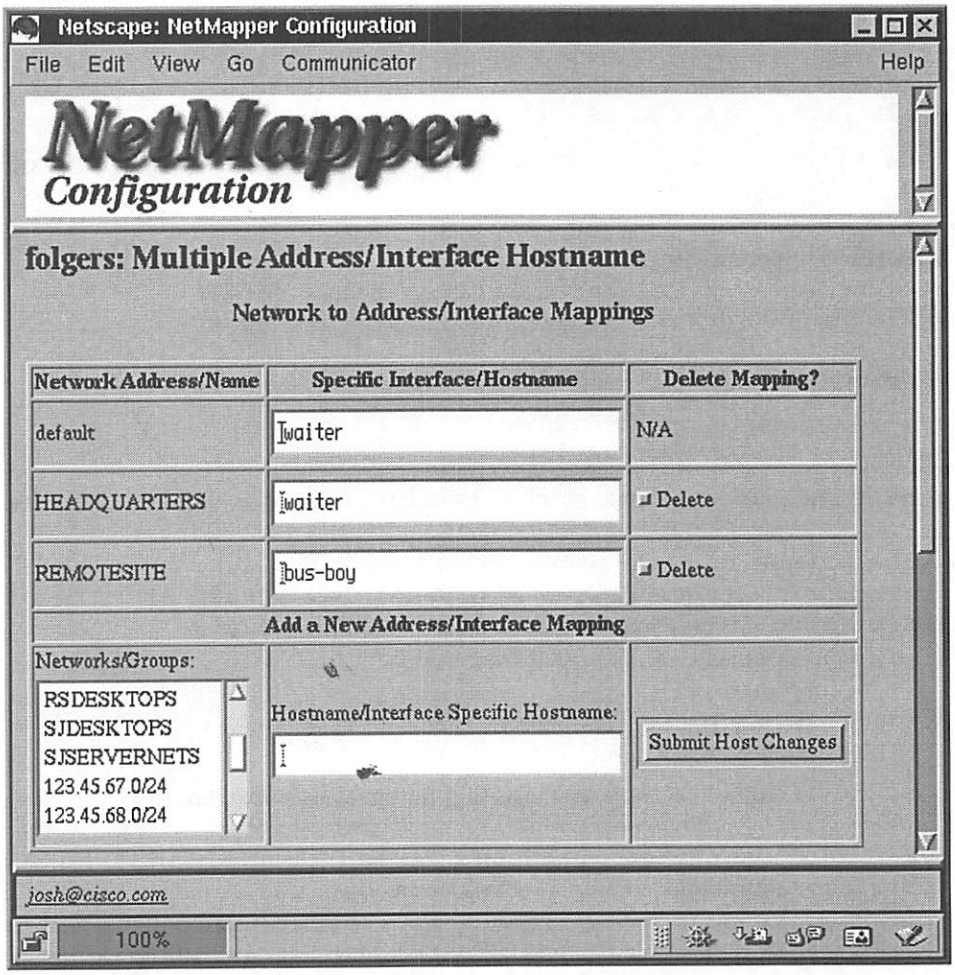


Figure 2: NetMapper configuration interface for sample host 'folgers'.

previously resolved. Thus, if a host address changed, the new address would never be known.) This interface allows the NetMapper administrators to change any information in the netmapper.conf file as well as providing some tools for testing and debugging.

Architecture/Internals

Version 1.0 of NetMapper is based on previous work along the same lines. This work was all done in Perl. The simple elegance of Perl hashes and powerful pattern matching kept Perl as the language of first choice for NetMapper. The previous work also used a Perl interpreted file to store configuration information. Availability of modules such as Data::Dumper to write Perl data structures out to a file made the decision to keep using this method a simple one.

Several data structures are used within the configuration file:

- A hash whose keys are the hostname 'aliases' which the NetMapper package is responsible for processing. The values of this top-level hash are anonymous references to other structures such as hash and array references. These other structures include such information as the default hostname or interface specific hostname, preferred hostnames based on network group names or CIDR format network addresses, available OS/architectural dependent partitions and possibly other information to be determined later.
- A hash representing the network hierarchy which consists of network IP addresses in CIDR format as well as symbolic names for groups of networks. The hash keys are either networks or network group names. The hash values are network group names of the parent of the key.
- A hash representing which OS's and machine architectures NetMapper knows about. NetMapper uses this data to produce the OS and architecture specific mount paths for indirect automounter map entries. An explanation of the OS/architecture specific partition naming is detailed in the NetMapper documentation.
- A scalar containing the version number of the NetMapper package/program (localmapper) which runs on the clients. (If the client program is not up to the current version, it can attempt to update itself via the network.)
- Various other pieces of information which would be maintained by the NetMapper administrator and should not be modified by NetMapper users (system administrators). This information might include interface names and error reporting email addresses.

NetMapper uses these structures to determine the desired hostnames for a network location in the following fashion:

1. localmapper runs on the client and initializes the NetMapper library which obtains configuration information as previously described.
2. NetMapper library functions examine network interfaces to obtain the client's active IP addresses and network lineage. (IP addresses

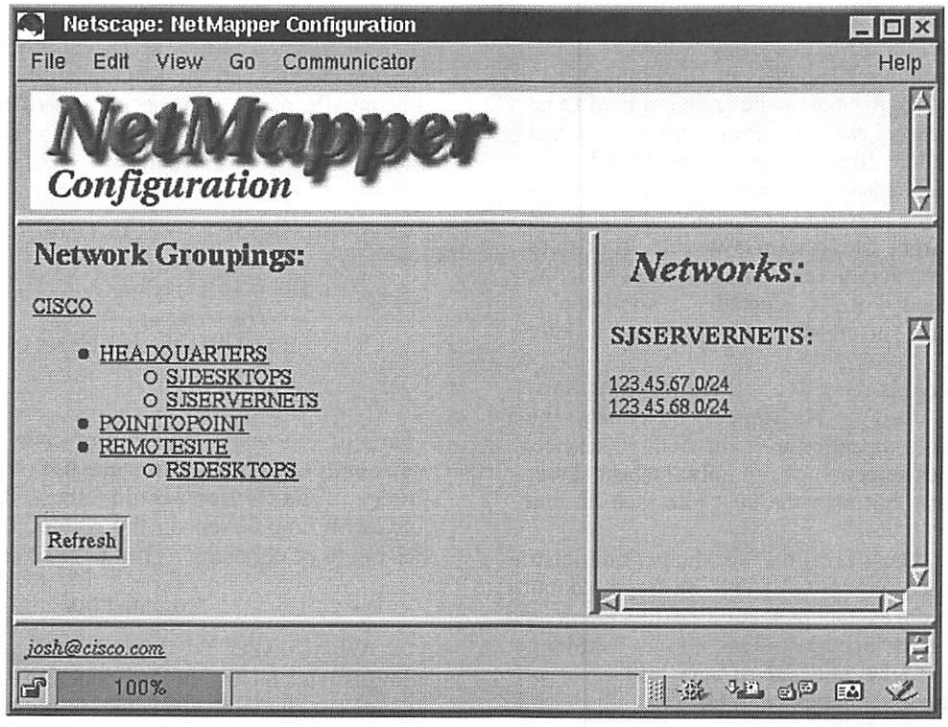


Figure 3: NetMapper network hierarchy displayed in configuration interface.

are obtained via standard UNIX commands such as netstat and ifconfig.) All CIDR format networks in NetMapper are reverse sorted by the number of significant network bits. The client's IP addresses are then compared to each element of the list of networks with a match occurring when the significant portion of the interface address and the network address are equal. The network lineage is then derived from the network hierarchy based on the matched networks. The lineage proceeds from most specific to most general network. Using the host 'rif-raf' from Figure 1 and the network hierarchy displayed in Figure 3, the lineage for 'rif-raf' would be as follows:

```
123.45.69.143/32
123.45.69.128/27
SJDESKTOPS
HEADQUARTERS
CISCO
default
```

The lineage (within the NetMapper network hierarchy) is derived via a NetMapper function (DeriveNetwork). This process yields a list of networks and network groups that this client belongs to.

3. For each hostname defined in NetMapper, a comparison is made between the lineage described above and the preferences defined in the NetMapper hostname structure. If no matches are found in the hostname definition the default will be used. If no default is found (or some other error occurs), the hostname alias will not be defined by NetMapper. (Thus hostname resolution for this entry would be determined through a NIS, NIS+ or DNS lookup.)
4. If the NetMapper hostname is determined to be an NFS server which serves architecture and OS dependent file systems, the preferred hostname will have additional match criteria applied – the preferred hostname must have the desired target file system available. If it does not, the first server candidate that does have the desired file system is selected. This fail-safe is a weak one and needs to be improved in later versions.
5. This process is recursive – If the resulting hostname is itself a NetMapper hostname, the lookup will repeat. For a distributed service, this assures selection of not only the best server for a client, but also the best interface on that server.

If part of this process fails, the NetMapper administrators are notified via email. The message contains information such as the hostname of the client and what type of error occurred. This allows NetMapper administrators to be notified of new unknown networks, machine architectures and OS's.

The NetMapper Perl module itself makes use of other modules available on the Comprehensive Perl Archive Network (CPAN) [19] such as Net::DNS and Net::Cmd. It's use and available functions can be viewed via perldoc or pod2text.

The NetMapper configuration server, nmconfd, is a very basic interactive script. It is currently written to be run via inetd. It supports a small set of commands and returns results in plain text. The command set and results are compatible with the Net::Cmd Perl module. Both reading and writing of netmapper.conf are protected by file locking to prevent reading a partially written file or simultaneous writes from the administrative interface.

The web configuration interface is not extremely complex. It makes extensive use of the CGI Perl module. It does rely on Javascript to work with frames and pop-up text lists of networks with certain mouse-overs.

Other Uses!

This paper has discussed hostname resolution in the context of modifying a client's local /etc/hosts file. It is important to point out that NetMapper is actually a Perl module. The localmapper script utilizes the NetMapper module to perform its host file manipulations, but this is only one possibility. In the following example, the NetMapper module is used to select a specific hostname in a script that will modify /etc/resolv.conf.

A systems administrator might want to populate client /etc/resolv.conf files with different nameserver addresses depending on network location. He or she could define some virtual NetMapper hostnames, perhaps something as creative as 'nameserver1' and 'nameserver2'. The new script could use the NetMapper module and call the appropriate function to return the specific real server hostnames of the appropriate 'nameserver1' and 'nameserver2' for the network(s) a client is on. The real code to lookup 'nameserver1' follows:

```
use NetMapper qw(:standard);
print
   ResolveHostname(
      'nameserver1',
      [BuildNetworkList()]
   ),
"\n";
```

That's it! Practically one line of code (plus the 'use' statement) is all it takes to make use of the NetMapper library. The script could then build a new /etc/resolv.conf based on the results. Other possibilities might be choosing a print or NTP server.

Conclusions

What makes NetMapper unique is that the administrator makes the decision about where network clients are directed via name resolution. NetMapper can be used to 'load balance' [20,21] among servers, but there are other products available that might do a

better job. NetMapper works best when decisions have to be made along some very 'human' aspect of computing - something an administrator knows that the computer can't easily figure out. A great example of this is license servers. Two groups might each have their own individual pool of licenses for the same product. Each group has their own license server. If the two groups can be split into distinct networks, an administrator could create one NetMapper virtual hostname – perhaps 'dmv'. The licensed software need only be told to look to a server named 'dmv' for its license. One group (of networks) would resolve 'dmv' to one server, while the other would use the second server. Users and administrators would not have to configure environment variables or 'dotfiles'.

Availability and Compatibility

The NetMapper module and localmapper script have been tested on SunOS 4, SunOS 5, HPUX 9 and HPUX 10. It is believed to work on IRIX, AIX, HPUX 11 and Linux. The Perl code requires Perl 5 or better. The Perl module and associated scripts (and whatever else I put in there...) are available in a tar/gzip file via anonymous FTP at ftp://ftpeng.cisco.com/josh/NetMapper.tgz.

Future Directions

The future of NetMapper depends on many things. (Like the results of this paper...) Without knowing what the future holds, I can only mention some of the things I have thought about changing.

Version 1.0 is not very smart about picking a secondary host if the first selection of an architecture and OS specific host does not have the desired target file system available.

The nmconfd Perl script should probably be made into a full-fledged binary daemon. It currently is slow due to being run by inetd and then compiled.

The algorithms NetMapper uses to determine the appropriate hostname could be re-written in 'C' or some other compiled language. (I keep hoping I can put this off long enough for the Perl compiler to reach maturity.) This way they could be incorporated into a renegade DNS server (if someone wanted to do such a thing). For a majority of cases, this would remove the client portion of NetMapper.

Acknowledgments

Lanny Ripple – for writing some of the central functions within the NetMapper module that perform the network matches.

Steve Bower – for convincing me early on that CIDR style network addresses would make things better and that Perl anonymous references were a 'good thing'.

Krish Sivakumar – for walking into my cube and saying something along the lines of "...why don't you just tweak the local hosts file?"

Mark Baushke – for letting me copy the method by which we split up architecture and OS specific partitions.

References

[1] RFC 819, *The Domain Naming Convention for Internet User Applications*, Zaw-Sing Su, Jon Postel. August 1982, http://www.isi.edu/in-notes/rfc819.txt.

[2] James M. Bloom, Kevin J. Dunlap, University of California, Berkeley, "Experiences Implementing BIND, A Distributed Name Server for the DARPA Internet," *Proceedings of the Summer 1986 USENIX Conference.*

[3] comp.protocols.tcp-ip.domains FAQ, Section 5, Question 5.14, *Order of returned records*, http://www.intac.com/~cdp/cptd-faq/section5.html#order.

[4] comp.protocols.tcp-ip.domains FAQ, Section 5, Question 5.24, *Different DNS answers for same RR*, http://www.intac.com/~cdp/cptd-faq/section5.html#diferentRR.

[5] RFC 1123, *Requirements for Internet Hosts, Application and Support*, Chapter 6. R. Braden, Editor, Internet Engineering Task Force, October 1989, http://www.isi.edu/in-notes/rfc1123.txt.

[6] RFC 1094, *NFS: Network File System Protocol Specification*, Sun Microsystems, Inc., March, 1989, http://www.isi.edu/in-notes/rfc1094.txt.

[7] Brent Callaghan, Tom Lyon, Sun Microsystems, Inc., "The Automounter," *Proceedings of the Winter 1989 USENIX Conference.*

[8] Cisco Systems, Inc., *Load Balancing: A Solution for Improving Server Availability*, http://www.cisco.com/warp/public/cc/cisco/mkt/scale/locald/tech/lobal_wp.htm.

[9] Cisco Systems, Inc., *LocalDirector in the Data Center*, http://www.cisco.com/warp/public/cc/cisco/mkt/scale/locald/tech/ldir_wp.htm.

[10] Kevin Delgadilloi, *Cisco IOS Product Marketing*, "Cisco DistributedDirector," http://www.cisco.com/warp/public/cc/cisco/mkt/scale/distr/tech/dd_wp.htm.

[11] Cisco Systems, Inc., *The Effects of Distributing Load Randomly to Servers*, http://www.cisco.com/warp/public/cc/cisco/mkt/scale/distr/tech/ddran_wp.pdf.

[12] Michael A. Cooper, University of Southern California, "Overhauling Rdist for the '90s," *Proceedings of the Sixth Systems Administration Conference* (LISA '92).

[13] Tom Christiansen, "Perl 5.0 Overview," USENIX *;login* magazine, Nov/Dec 1993, Volume 18, http://www.usenix.org/publications/login/christiansen.html.

[14] Free Software Foundation, *m4 Online Manual*, http://www.fsf.org/manual/m4/index.html.

[15] RFC 1517, *Applicability Statement for the Implementation of Classless Inter-Domain Routing*

(CIDR), Internet Engineering Steering Group, R. Hinden, September, 1993, http://www.isi.edu/in-notes/rfc1517.txt.

[16] RFC 1518, *An Architecture for IP Address Allocation with CIDR*. Y. Rekhter, T. Li, September, 1993, http://www.isi.edu/in-notes/rfc1518.txt.

[17] RFC 1519, *Classless Inter-Domain Routing (CIDR): An Address Assignment and Aggregation Strategy*, V. Fuller, T. Li, J. Yu, K. Varadhan, September, 1993, http://www.isi.edu/in-notes/rfc1519.txt.

[18] RFC 1520, *Exchanging Routing Information Across Provider Boundaries in the CIDR Environment*, Y. Rekhter and C. Topolcic, September, 1993, http://www.isi.edu/in-notes/rfc1520.txt.

[19] *Comprehensive Perl Archive Network (CPAN)*, http://www.cpan.org.

[20] Internet Software Consortium, *DNS, BIND and load balancing*, http://www.isc.org/view.cgi?/products/BIND/docs/bind-load-bal.phtml.

[21] Roland J. Schemers, III, SunSoft, Inc., "lbnamed: A Load Balancing Name Server in Perl," *Proceedings of the Ninth System Administration Conference* (LISA '95).

Related Papers and Information

Cheng-Zen Yang, Chih-Chung Chen, and Yen-Jen Oyang, Department of Computer Science and Information Engineering, National Taiwan University, Taipei, "Clue Tables: A Distributed, Dynamic-Binding Naming Mechanism," *USENIX Summer 1994 Technical Conference*.

Chad Yoshikawa, Brent Chun, Paul Eastham, Amin Vahdat, Thomas Anderson, David Culler, University of California, Berkeley, "Using Smart Clients to Build Scalable Services," *Proceedings of the USENIX 1997 Annual Technical Conference*.

Giray Pultar, "Automatically Selecting a Close Mirror Based on Network Topology," *Proceedings of the 12th Systems Administration Conference*, (LISA '98).

Enhancements to the Autofs Automounter

Ricardo Labiaga – Sun Microsystems, Inc.

ABSTRACT

An automounter is a system tool that enables administrators to share a uniform file system name space across an organization. Until the introduction of the enhanced autofs automounter, no automounter had integrated browsability of maps into the operating system. This paper describes enhancements made to the autofs automounter that enable entry listing in automounter maps without triggering mount storms. This allows applications to seamlessly browse potential mountable entries without the overhead of file system mounts. In addition, this paper describes the implementation of lazy mounting of hierarchies, which improves on-demand file system mounting.

Introduction

An automounter is a service that automatically mounts file systems upon reference and unmounts the file systems after a period of inactivity. This service has been traditionally used to access NFS file systems in large enterprise environments where centralized administration of the file system name space is preferred. An automounter will perform the mounts when a file system is first referenced without requiring the workstation user to acquire super-user access to perform the mount command. To the user, the mount of the newly accessed file system is transparent.

Three automounters are in wide use today. The first to become available was the automounter in SunOS 4.0. This automounter is available in many Unix system platforms, such as those from Auspex, HP, Compaq, IBM and SGI. A second automounter, the Amd automounter [8], is included in 4.4 BSD and has also been ported to many Unix systems. The third, the autofs automounter, was introduced in 1993 in Sun Solaris 2.3 and has been implemented on various other Unix system platforms since then, such as HP, IBM, Linux and SGI.

Both the SunOS 4.0 automounter and the Amd automounter are implemented as NFS servers [4]. The server is a daemon process, NFS mounted on directories that are required to be dynamically mounted. Each mount point is associated with a map that determines what components appear under the mount point and describes to which file systems they correspond. When the user process crosses the mount point, the kernel communicates with the daemon in the same manner it would communicate with any other NFS server. The daemon mounts the desired NFS file system on an alternate path and returns a symbolic link to this newly mounted file system. All lookups crossing the mount point are redirected by the kernel to the NFS server daemon, which returns a symbolic link to the mounted NFS file system.

The autofs automounter is considerably different than the traditional NFS server based automounter. It consists of three main components; the autofs file system, an auxiliary automount command utility and the automountd daemon.

The autofs file system is a virtual file system [6] (VFS) that intercepts requests to access directories that are not yet present. It calls the automountd daemon to mount the requested directory. The automountd daemon locates the requested path in the automounter maps, mounts the corresponding file system overlaying the autofs mount point, or mounts it on a subdirectory within the autofs file system. The subdirectory is created if necessary. Once the requested file system has been mounted, the original operation which accessed the autofs directory can proceed. Subsequent references to the mounted file system are redirected within the kernel by the autofs file system. No further intervention of the automountd daemon is required.

It is the responsibility of the automount auxiliary command to initially install the autofs mounts that connect the automounter maps into the file system name space. At system startup, the automount command reads the auto_master map and installs the initial set of autofs mounts into the file system name space.

```
# Master map for automounter
#
/net    -hosts      -nobrowse
/home   auto_home
```

Table 1: auto_master map.

Given the auto_master map listed in Table 1, the automount command installs the -hosts map on the /net directory and the auto_home map on the /home directory. The -hosts map is a special automounter map consisting of all available hosts in the network along with their respective exported file systems. Figure 1 illustrates the resulting autofs file systems. The following would be the equivalent autofs mounts:

```
$ mount -F autofs -o nobrowse \
                    -hosts /net
$ mount -F autofs auto_home /home
```

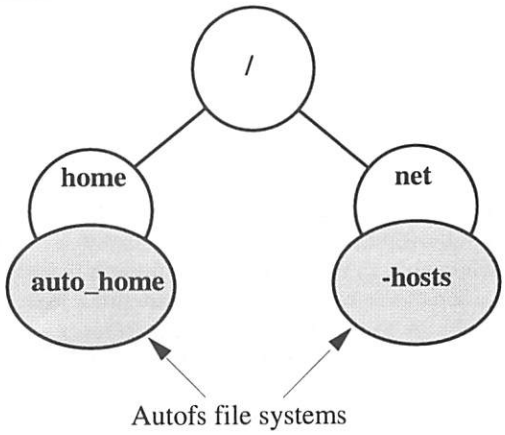


Figure 1: Autofs mount points.

By default the automountd daemon and the automount command will look for the maps in the /etc directory, unless otherwise specified in the auto_master map. The name service is consulted depending on the system configuration. On a Sun Solaris OS, the automount entry in /etc/nsswitch.conf determines whether or not the automounter will consult the name service directory. The current release of the Sun Solaris OS supports NIS and NIS+ [7] automounter maps. Support for automounter maps via the Lightweight Directory Access Protocol [5], will be added in a future release.

An autofs mount contains the following information:

- The name of the map associated with the mount point.
- The type of the map, either direct or indirect.
- The address of the user-level automountd daemon to contact for file system resolution.
- The default options to be used for mounting.

There are three kinds of automounter maps: direct, indirect, and executable. A direct map contains entries consisting of an absolute path, used as the key identifier, a corresponding file system to be mounted upon reference of the key and an optional set of mount flags to be used. Each direct map entry is itself installed as an autofs mount. Each one of these mounts constitutes a *direct mount*. A direct mount is defined as the autofs mount of an absolute path listed in a direct autofs map. With a direct mount, once the autofs mount point is accessed by a process, the autofs file system calls the automountd daemon providing its path and direct map name. The automountd daemon proceeds to mount the corresponding file system covering the autofs mount point. The autofs file system then redirects the blocked request to the newly mounted file system. Table 2 illustrates two direct map entries part of an auto_direct map.

```
/usr/dist -ro flash:/export/dist
/opt/onbld -ro flash:/export/onbld
```

Table 2: auto_direct map.

An indirect map contains entries consisting of a key identifier (a simple path component), a corresponding file system to be mounted upon reference of the key and an optional set of mount flags to be used. The indirect map itself is installed as the autofs mount, in contrast to direct maps whose entries are the actual autofs mounts. An *indirect mount* is defined as the autofs mount of an indirect map. With indirect mounts the autofs file system provides access to a directory of automatic mounts. Access to the autofs mount point itself does not trigger mounts, instead a request to lookup a subdirectory (key) in this directory will cause the autofs file system to call the automountd daemon with the subdirectory name and the map name. The daemon finds the name in the map, mounts the corresponding file system using the subdirectory as the mount point (not the autofs mount point) and replies to the autofs file system, which in turn redirects the blocked request to the new file system mounted. Table 3 illustrates an indirect map containing various home directory entries.

```
ashok   redback:/export/home/ashok
bev     turbo:/export/home/bev
brent   terra:/export/home/brent
david   jetsun:/export/home/david
warp    hp:/export/warp
peter   turbo:/export/home/peter
spencer austin:/export/home/spencer
```

Table 3: auto_home map.

An executable map is a special kind of indirect map. It is a local file with its execute bit set. The automountd daemon will execute the file and provide the name of the key to be looked up as an argument. The executable map returns the corresponding map entry on its stdout or no output if the entry cannot be determined. Refer to [3] for a detailed description of executable maps.

An extensive description of the features of the various automounters can be found in the references [1, 2, 3, 4, 8] at the end of this paper.

Limitations of the Autofs Automounter

Although the autofs automounter solved a number of problems seen in previous automounters [3], some limitations remained:

- There was inherent serialization in the autofs file system kernel component. This prevented concurrent mounts of entries that were part of the same automounter map.
- No support for browsability of maps was provided. A listing of an autofs directory only returned entries that had previously been mounted. Entries that had not yet been mounted were not listed.

- The mechanism used to mount and unmount hierarchies of file systems was prone to error. Temporary outage of a component lead to holes in the file system name space.

Architecture

In order to fix these limitations, the autofs automounter was re-architected for Sun Solaris 2.6. The new version remains split into three distinct components:

- The autofs file system, a kernel virtual file system that triggers all of the mounting and unmounting of file systems.
- The automountd daemon, a user level process responsible for performing the actual mounts and unmounts of the requested file systems.
- The automount command, a user level program that installs the initial autofs file system entry points.

The autofs file system and the automountd daemon communicate using connection-oriented RPC over the loopback transport. The RPC Protocol section describes this protocol in detail.

Browsability of Indirect Maps

Until the introduction of the enhanced autofs automounter, automounters lacked the ability to list all available entries in maps. A listing of an autofs directory referencing an indirect map only returned a list of directories already mounted, but did not list the entries which could potentially have been mounted, making it difficult for the end user to know what entries were available before they were first accessed.

This restriction was in place to prevent unintentional mounts of the entire map contents, a condition known as a *mount storm*. For instance an ls -l /home/* command, would have triggered a mount storm. This mount storm would have been caused by the *readdir*(3C)/*stat*(2) combination. The *readdir*(3C) makes the entire list of keys available to the browsing utility, such as *ls*(1) or a GUI file manager, which in turn issues a *stat*(2) of every entry to obtain its attributes. The *stat*(2) of an indirect map entry would have triggered the mount of the entry in order to verify its existence, and obtain its attributes. Performing this operation on every entry would have made directory browsing expensive and time consuming on relatively large maps.

The enhanced autofs automounter solves the mount storm problem by modifying the mount strategy. Previous automounters trigger the mount as soon as a new entry is looked up in an automounter map, so stat(/home/user) triggers the mount of the file system that corresponds to the user entry in the auto_home map. The enhanced autofs automounter postpones the mount of the file system until the indirect map entry is opened or a lookup of a component underneath it takes place. A simple *stat*(2) of the entry does not trigger a mount, instead the automountd daemon will simply query the map for existence of the entry.

Indirect maps may include a wildcard key "*", which tells the automountd daemon that any key is valid in the map. In such cases, since not all entries are explicitly defined in the map, a listing of the map needs to be the combination of the dynamically created entries (those previously matched and mounted via the wildcard key) and the explicitly defined entries. This is achieved by first listing the entries that have already been mounted (either explicitly or via the wildcard) and then listing all explicitly defined entries in the map. A search for duplicates is done by the autofs file system before returning from the *getdents*(2) system call.

The first *getdents*(2) of an indirect mount point will return the entries currently mounted. Once all mounted entries have been listed, subsequent *getdents*(2) cause the autofs file system to send an AUTOFS_READDIR request to the automountd daemon to list the remaining entries in the map. This is done repeatedly until all entries in the map have been listed. The name of the map, the offset within the directory, and the number of bytes requested are included in the request. The automountd daemon will return the directory entries in chunks of the requested size, starting at the requested offset. Duplicates of entries already mounted are filtered out by the autofs file system. The maximum number of mounted entries listed is specified by the value of AUTOFS_DAEMONCOOKIE. This value is OS specific, though is usually in the order of tens of thousands of entries.

Consider the auto_home map listed in Table 3. Assume that the /home/brent and /home/peter directories have previously been accessed and mounted, and that our *readdir*(3C) library function generates *getdents*(2) requests of 25 bytes. A call to readdir(/home) will generate the following sequence of *getdents*(2) operations:

- The first getdents(/home, 25) lists the previously mounted entries brent and peter, and sets the next offset to AUTOFS_DAEMONCOOKIE to indicate it is done listing all currently mounted entries.
- Since the end of the directory has not yet been reached, a second getdents(/home, 25) is issued. Given that the current value of the offset is AUTOFS_DAEMONCOOKIE, the autofs readdir code handler issues a request to the automountd daemon, requesting the next 25 bytes in the directory. The automountd daemon in turn returns the first four entries in the map (22 bytes including termination characters) ashok, bev, brent, and david, and sets the next offset to AUTOFS_DAEMONCOOKIE + 4, since it returned four entries. It is the autofs file system that filters out the brent entry, since it is currently mounted and has already been listed.
- Again, since the end of directory has not yet been reached, another getdents(/home, 25) call is issued. The request is sent to the automountd

daemon again, which then returns the warp, peter, and spencer entries. The automountd daemon sets the end of directory boolean indicator in the structure containing the result, and replies to the autofs file system. The autofs file system filters out the peter entry since it is currently mounted, and *getdents*(2) returns the remaining two entries to the caller.

To maximize performance, and minimize kernel memory usage, no nodes are created as a result of a *getdents*(2) request. Nodes for the given entries will only be created if a process makes a subsequent lookup of entries listed by *getdents*(2) [i.e., ls -l].

These nodes are created with default attributes, which causes *stat*(2) to report the default attributes of nodes on which no file system has yet been mounted, instead of the directory attributes of the root of the mounted file system. Once the real mount is triggered, the attributes of the root of the mounted file system will be reported. There is no way to obtain the true attributes of the root of the file system without mounting the file system first. Remember that the intention is to be able to list the contents of an automounter map without having to mount the file systems referred to by the entries in the map. Note that this is the same behavior of direct map entries in previous automounters.

Performance

When an AUTOFS_READDIR request is received by the automountd daemon, it issues a request to the name service for the contents of the requested map. Since name services such as NIS and NIS+ do not support requests using byte ranges, the entire map needs to be requested on the first AUTOFS_READDIR call, even when only a portion of the map entries will be returned to the autofs file system at a time. Since the automountd daemon caches the entry listing, a subsequent AUTOFS_READDIR invocation will obtain its information from this cache. This leads to good performance of relatively large automounter map listings.

The directory listing of a 13,000 entry NIS automounter map (ls -f) takes approximately 15 seconds on a cold cache and two seconds on a warm cache. A long directory listing, which enumerates entries and their corresponding attributes (ls -l), takes approximately 40 seconds. A listing of a map with just under 150 entries is practically instantaneous. These informal measurements were obtained on a single CPU Sun Ultra 1 workstation running at 167 Mhz over a 10 Mb ethernet.

Optimizing Browsability of Maps

Information is more readily available if it is hierarchically organized. For example a large corporation may be divided into divisions and each division may be subdivided into organizations. Members of a team are easily identifiable when the division and organization to which they belong is known.

The autofs file system name space can be organized in a similar manner. It is easier to browse and administer multiple small automounter maps, than a single large flat map. Each automounter map can contain a combination of autofs and non-autofs mounts. For example, the company previously mentioned makes its employee home directories available through a hierarchical name space that mirrors its organizational structure. Its home directories are rooted at /home and the auto_master map contains the single entry:

```
/home    auto_home
```

The auto_home map contains:

```
legal    -fstype=autofs  auto_legal
eng      -fstype=autofs  auto_eng
HR       -fstype=autofs  auto_HR
mktg     -fstype=autofs  auto_mktg
CEO      flash:/export/olsen
```

The auto_eng map contains:

```
OS       -fstype=autofs  auto_OS
AMI      -fstype=autofs  auto_AMI
java     -fstype=autofs  auto_java
VP       flash:/export/smith
CTO      flash:/export/jackson
```

The auto_OS contains:

```
mct      iceberg:/export1/mct
spn      iceberg:/export1/spn
jim      flash:/export/jim
...
```

The user jim accesses his home directory as /home/eng/OS/jim. This results in the autofs mount of the auto_home, auto_eng, and auto_OS maps, as well as the NFS mount of flash:/export/jim. A directory listing of jim's organization simply returns the contents of the auto_OS map.

Potential Problems

Even though in practice mount storms are very rare, they can still occur. A recursive listing of entries under an indirect mount point can trigger the mount of all the file systems in the map. This may prove to be expensive and time consuming. It is conceivable that some legacy scripts and applications will need to be fixed to avoid this situation, especially those that do a depth-first search of the automounted file system at any point in order to build their own cache.

Disabling Browsability

Browsability may be disabled through the use of the -nobrowse map option for a specified map and children maps. The option is inherited. This option instructs the autofs file system not to list the contents of the specified map, causing it to simply return the listing of entries already mounted. This is similar to the old behavior, where only mounted entries were listed. The system administrator may want to specify this option in the rare event that a given application causes mount storms or when maps are unreasonably

large, causing browsing to negatively affect system performance. This option only applies to the specified map and does not affect browsability on other areas of the file system name space. For instance, Table 1 shows the /net autofs mount point installed with the -nobrowse option. A *readdir*(3C) of the /net directory will only list the entries that have already been mounted. It will not list potentially mountable entries. The -browse option is used to re-enable browsability at any point in the autofs hierarchy. The default is -browse.

Incompatibility Issues

The -nobrowse and -browse options can not be parsed by older automounters. These options are not intended for inclusion in maps accessed via the shared name services, unless all automounters understand the new options or simply ignore unknown options. If an older automounter accesses an entry containing either of these options (or any other unknown option for that matter), it will cause the mount to fail since the older automounter cannot correctly parse the new options. The local system automounter map files are a better location for these options.

Improved Concurrency

The autofs automounter concurrency was severely limited when it came to accessing multiple automatic mounts under an indirect autofs mount.

In the case of multiple concurrent accesses to a single direct autofs directory, only one request needs to be issued to the automountd daemon to mount the corresponding file system. All other requests should block for the originating request to finish. This was achieved by flagging the autofs mount point with a "mount in progress" flag. Once the originating request received its reply from the automountd daemon, it signaled all other threads waiting on this mount point to proceed and clear the "mount in progress" flag. No new requests needed to be made to the automountd daemon since the mount had already been installed.

In the case of indirect mounts, the autofs directory was not the mount point of the newly mounted file system. Instead, the automountd daemon created a new mount point as a subdirectory of the autofs directory and mounted the new file system on top of it. At the time the request to the automountd daemon was generated, the autofs file system had no unique mount point to use as a synchronization point. Instead, it used the autofs indirect mount point to block other threads from generating a second mount request for the subdirectory. Unfortunately, this had the side effect of blocking lookups and mounts of other subdirectories under the same autofs indirect directory. This affected concurrency and lead to unnecessary hangs, where the lookup of a given path needed to wait for the mount of an unrelated path to be finished by the automountd daemon.

For instance, access to /home/warp first flagged the /home node with the "mount in progress" flag and then requested the automountd daemon to mount the corresponding file system on /home/warp. This is illustrated in Figure 2.

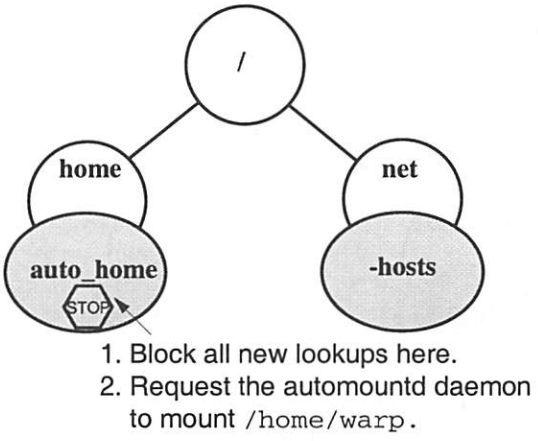


Figure 2: stat(/home/warp).

The automountd daemon in turn created the warp directory under the /home mount point and issued the mount system call to mount the new file system. If, at the same time, access to /home/bev was requested by a second process, the second process would unnecessarily block until the first process' access completed, since the /home node had previously been locked. Having a multithreaded automountd daemon was of no particular use at this point, since the request to mount /home/bev was blocked by the autofs file system. Figure 3 illustrates this scenario.

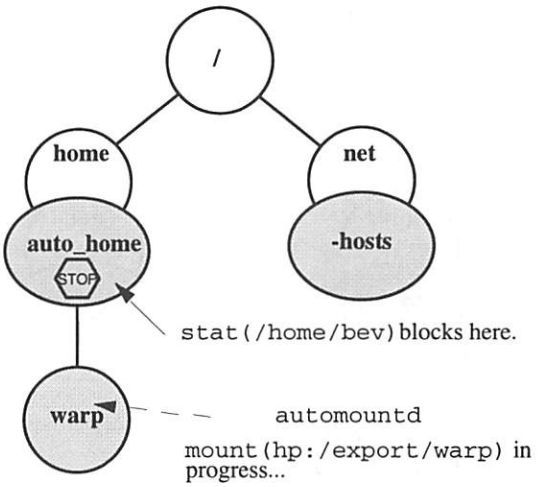


Figure 3: stat(/home/bev).

Once the mount of the new file system on /home/warp finished, the second request to install the /home/bev mount could be made to the automountd daemon.

To fix this major limitation, the enhanced autofs automounter moves the creation of the mount point

from the automountd daemon to the autofs file system. For instance, upon lookup of /home/warp, the autofs file system determines that warp has not previously been accessed, creates the node for the name, warp, flags it with a "lookup in progress" flag, and issues a lookup request to the automountd daemon, as illustrated in Figure 4.

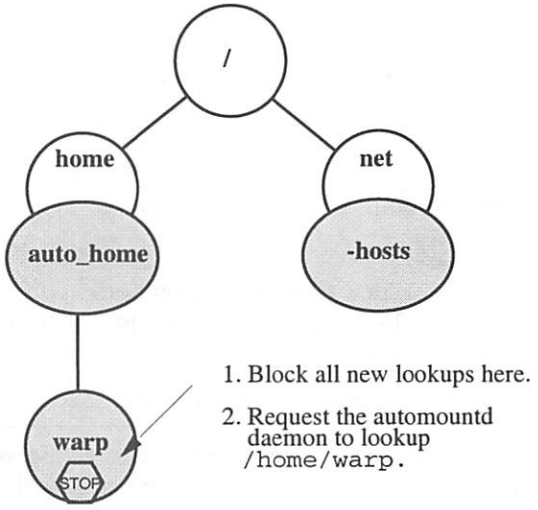


Figure 4: Non-blocking stat(/home/warp).

The daemon looks up the warp entry in the map and returns either success or failure to the autofs file system. A concurrent lookup of /home/bev can proceed since the /home node is not locked. The autofs file system creates a new node for the name, bev, under /home, sets the "lookup in progress" flag on /home/bev and sends the request to the automountd daemon. This is illustrated in Figure 5.

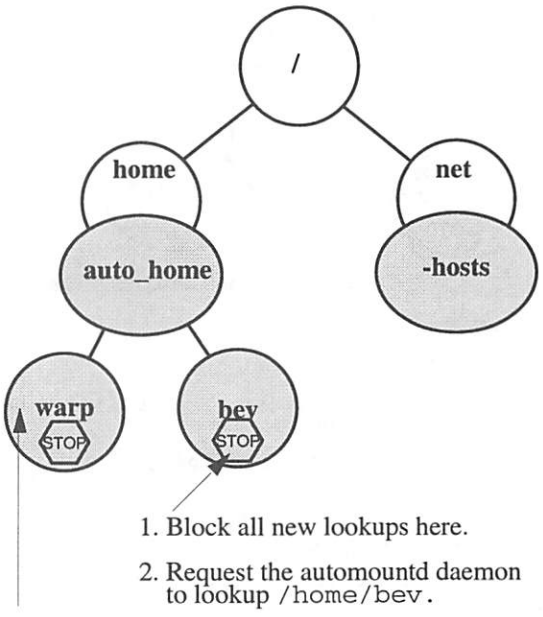


Figure 5: Concurrent stat(/home/bev).

At this time all new concurrent accesses to /home/bev will be blocked until the automountd daemon replies, since the bev node is flagged with "lookup in progress." This prevents flooding the automountd daemon with requests to perform duplicate lookup work. When the calling thread receives its reply from the automountd daemon, it will wake up any other threads waiting on the bev node, at which time they can return the same lookup status as the calling node obtained from its request to the automountd daemon.

Thus far, the nodes have only been looked up, no mounts have taken place. In order for the node to be covered with its corresponding file system, the node needs to be opened, or a lookup of a component underneath the node needs to take place. Section 3 describes the mount trigger policy along with the autofs RPC communication protocol in more detail. After the automountd daemon is done mounting the file system, the synchronization flags on the mount point are cleared and every process is allowed to proceed with its lookups and traverse into the newly mounted file system. This is illustrated in Figure 6.

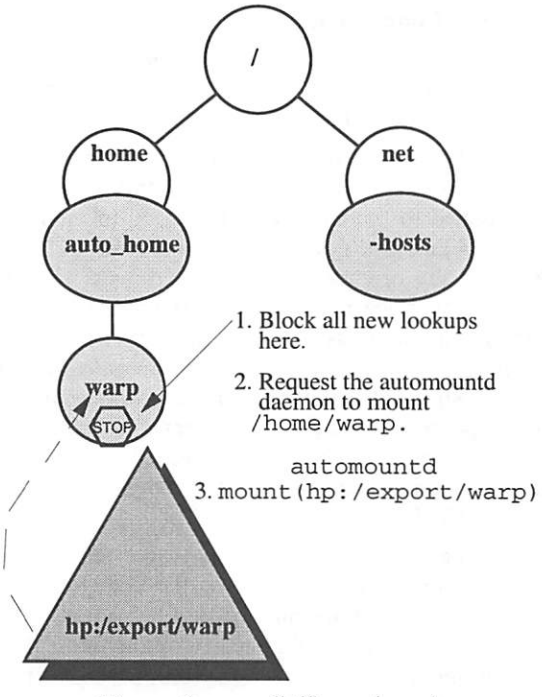


Figure 6: opendir(/home/warp).

Lazy Mounting of Hierarchies

Many times, more than a single file system needs to be mounted as part of a file system hierarchy. Existing automounters mount all members of this file system hierarchy at once, as soon as the top level file system is accessed. A common example of this is the use of the /net autofs mount. The automounter mounts all file systems from a given server when /net/host is referenced. These file systems may be hierarchically related, one within the other, and the entire hierarchy

is mounted as a unit. As a side effect, the entire hierarchy of file systems needs to be unmounted as a unit as well. This presents a number of problems. First, a large number of file systems are mounted, even if only one is needed. Second, and a more significant problem, is that nested file systems have to be recursively unmounted and remounted if the entire hierarchy can not be unmounted. The remounting of elements is particularly prone to failure due to network discontinuity and time outs, which often result in a hole in the file system name space.

To address this problem, the enhanced autofs automounter mounts only the top level file system of a hierarchy on first reference, installing autofs trigger nodes where the next level file systems will need to be mounted upon reference. This provides better on-demand automounting of hierarchies by reducing the total number of mounts. The autofs trigger nodes are direct mount points. They trigger a request to mount the real file system when the trigger node is opened or a lookup of a component under this directory is performed.

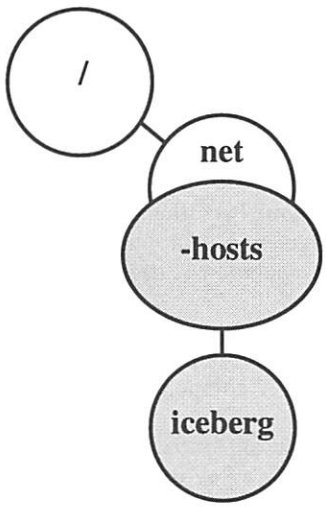


Figure 7: stat(/net/iceberg).

The automatic unmounting process is now reduced to unmounting the top level file system and its trigger nodes, instead of *n* file systems, multiple levels deep. If the top level file system is busy, only the trigger nodes are remounted, which is considerably faster than remounting other file systems, since the remount does not involve context switching in user space or over-the-wire communication with potentially dead or slow servers.

The following example describes the new mechanism for mounting hierarchical mounts in the enhanced autofs automounter. Assume the server iceberg exports the following eleven file systems:

```
/
/export1
/export1/home
/export2
/export3
...
/export9
```

/net is an autofs mount point that references the -hosts map.

As illustrated in Figure 7, a lookup of /net/iceberg creates the new node and makes a call to the automountd daemon requesting that it lookup the entry. If such entry exists, the new node is returned. If the entry does not exist, the node is removed and an ENOENT error is returned.

An opendir of /net/iceberg triggers a call to the automountd daemon to NFS mount iceberg:/, the top-level file system. After this top-level file system has been successfully mounted, the automountd daemon replies to the autofs file system indicating it needs to install nine new trigger nodes located at:

```
/net/iceberg/export1
/net/iceberg/export2
...
/net/iceberg/export9
```

Notice that only the trigger nodes for the level immediately following the top level were installed. This is illustrated in Figure 8.

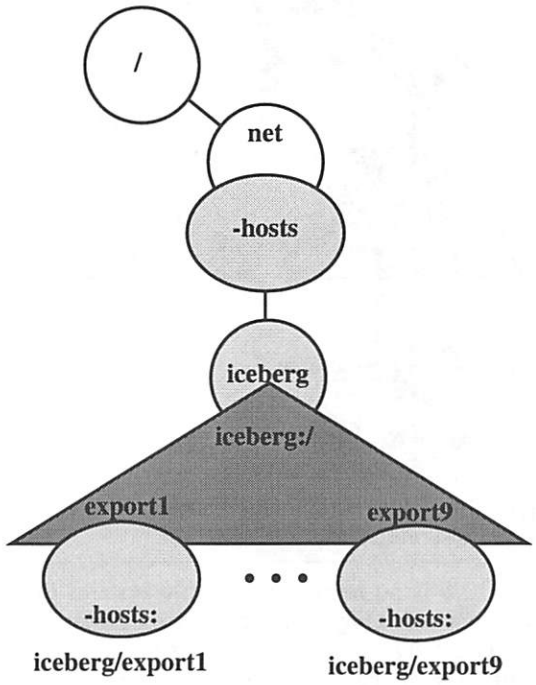


Figure 8: Top-level and trigger nodes.

If export1 is later referenced, the NFS file system iceberg:/export1 will be mounted, along with a trigger node for the next level of mounts, /export1/home in this case. This is illustrated in Figure 9. A subsequent access of /net/iceberg/export1/home would trigger the NFS mount of iceberg:/export1/home. Notice that no triggers under /net/iceberg/export1/home need to be installed, since the server does not share any file systems rooted deeper than iceberg:/export1/home.

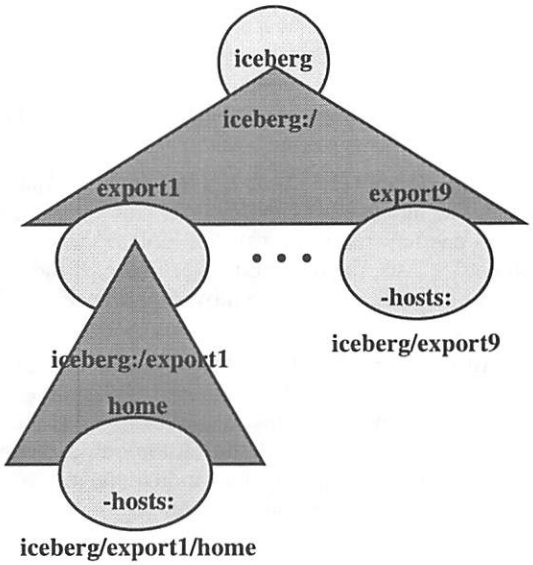


Figure 9: stat(/net/iceberg/export1/dir).

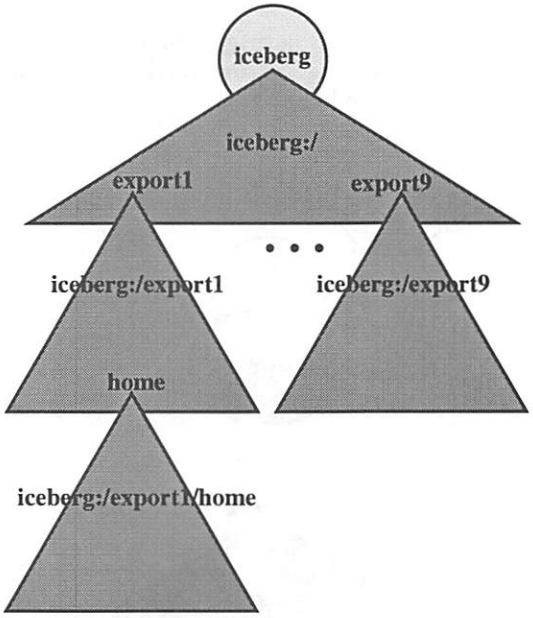


Figure 10: Older automounters mount all file systems on first reference.

There is no need to mount file systems that are never touched. Only the file systems accessed have been mounted along with the trigger nodes that enable the mounts of the next level. In contrast, the older automounters would have mounted the entire hierarchy on first reference of /net/iceberg as depicted in Figure 10.

Unmounting

The automatic unmount of a hierarchy is performed depth-first. A file system and its trigger nodes are unmounted as a unit. In the example depicted in Figure 9, the iceberg:/export1 file system and the iceberg/export1/home trigger node are both unmounted at the same time. This leaves the iceberg/export1 trigger node exposed, ready to trigger a new mount if a future access takes place. The unmount of iceberg:/ will not be attempted as long as any of its trigger nodes are covered by a mounted file system.

The RPC Protocol

The autofs file system and the automountd daemon communicate using connection-oriented RPC over the loopback transport. The communication protocol has been extended to six basic operations. All RPC calls are initiated by the autofs file system, they wait in a listening state until the automountd daemon replies or the calling thread is interrupted. The autofs file system will initiate a call to the automountd daemon when a thread references a trigger autonode. The six basic operations follow:

AUTOFS_LOOKUP

Description: Performs a lookup of a key in a map. This request is triggered when a thread performs a lookup of a not-yet mounted key in an indirect autofs mount.

Arguments: The request takes as argument the name of the map, the key being looked up, and the directory path.

Results: The automountd daemon returns success if the entry exists in the map or an error if it does not exist.

AUTOFS_MOUNT

Description: Performs a mount of a key from a map at the given mount point using the specified options. This operation is triggered under the following conditions:

- Direct autofs mounts: Triggered when a thread opens a direct autofs mount or performs a lookup of a component of the direct autofs mount.
- Indirect autofs mounts: Triggered when a thread opens a key of an indirect autofs mount (i.e., opendir(/home/warp)) or performs a lookup of a component below the key of an indirect autofs mount (i.e., stat(/home/warp/.cshrc)).

The automountd daemon matches the corresponding entry in the map and performs the required mount. For hierarchical mounts, the automountd daemon builds a list of structures corresponding to the next level mounts, each containing the necessary information to perform an autofs mount where a new file system mount needs to be triggered. These are used by the autofs file system for installation of the next-level trigger nodes. Certain types of map entries do not require a top-level mount. For these entries, the automountd daemon does not perform any mounts itself and simply returns a list of mount structures to be installed by the autofs file system as trigger nodes.

Arguments: The request takes as argument the name of the map, the key being mounted, the mount point, the default map options, and the type of the map.

Results: The automountd daemon returns a status code indicating the result of the operation and any necessary trigger nodes that need to be installed by the autofs file system.

AUTOFS_POSTMOUNT

Description: Adds a mount entry to etc/mnttab. Many Unix implementations maintain the mount table in user space (/etc/mnttab). To facilitate access to this table, an AUTOFS_POSTMOUNT operation is provided. A request is sent to the automountd daemon to append a path to /etc/mnttab after an autofs in-kernel mount has been performed. This is not necessary for OS implementations that maintain an in-kernel mount table.

Arguments: The request takes as argument a list of special devices, their mount point, file system type, mount options, and device identifiers to be added to the mount table.

Results: The automountd daemon returns a status code indicating success or failure.

AUTOFS_UNMOUNT

Description: Unmount the path corresponding to the specified device identifier.

Arguments: The request takes as argument the device identifier of the file system to unmount.

Results: The automountd daemon returns a status code indicating success or failure.

AUTOFS_POSTUNMOUNT

Description: Remove the path from /etc/mnttab corresponding to the specified device identifier. This is used to complete unmounts performed in the kernel (not via the *umount*(2) system call). Not required for OS implementations that maintain an in-kernel mount table.

Arguments: The request takes as argument a list of device identifiers to be removed.

Results: The automountd daemon returns a status code indicating success or failure.

AUTOFS_READDIR

Description: Requests a list of entries in a map. The automountd daemon reads these entries from the specified map and returns an array of *dirent*(4) structures beginning at the specified offset.

Arguments: The request takes as argument the name of the map, the starting offset and the total number of bytes requested.

Results: The automountd daemon returns an array of *dirent*(4) structures, the total size of the entries read, the last offset in the list, and an end of directory indicator.

Availability

The enhanced autofs automounter is available in Sun Solaris 2.6 and Solaris 7. The source is available to all ONC+ licensees, which include many major Unix vendors.

Future Work

An important limitation of automounters today (including the enhanced autofs automounter described on this paper) is that changes to the automounter maps are not effective immediately if the modified map entry has previously been mounted. For instance, if a given server shares a new file system after a client has already mounted a previously shared file system, the client will not have access to the new information. The newly exported file system will become visible to the client after /net/server and its trigger nodes are automatically unmounted after a period of inactivity on the client and mounted again on subsequent access. This is problematic to the user since newly exported information can not be readily accessed on clients.

Clearly, one solution to this problem is to provide some kind of mechanism that notifies the autofs file system that the mounted hierarchy has changed. However, not all directory name services provide time stamp capabilities, which would make it difficult for the automounter to determine when the maps have been updated. Some of the automounter maps are dynamic, such a -hosts. It is not practical to continuously poll servers to see if their list of shared file systems has changed. One solution would be a mechanism to manually notify the autofs file system with a list of mount points that need to be refreshed. A new option to the automount command specifying the path that needs to be refreshed may suffice.

Acknowledgments

The author would like to thank his colleagues from Sun Microsystems, Inc. who have provided valuable ideas and discussion material for the enhancement of the autofs automounter, in particular Ashok Advani, Brent Callaghan, David Robinson, and Peter Staubach.

Author Information

Ricardo Labiaga is a Member of Technical Staff at Sun Microsystems, Inc. He holds a Bachelors of Science degree in Computer Science and a Masters of Science in Computer Engineering from The University of Texas at El Paso. For six years, he has worked on a variety of projects within the Network File Systems Group at Sun Microsystems, Inc., with a primary focus on automounting. Reach him electronically at <labiaga@eng.sun.com>.

References

[1] Brent Callaghan, "The Automounter – Solaris 2.0 and Beyond," *1992 Sun User Group Conference Proceedings*.

[2] Brent Callaghan, "The Automounter – Using it Effectively," *1990 Sun User Group Conference Proceedings*.

[3] Brent Callaghan, Satinder Singh, "The Autofs Automounter," *Summer 1993 Usenix Conference Proceedings*.

[4] Brent Callaghan, Tom Lyon, "The Automounter," *Winter 1989 Usenix Conference Proceedings*.

[5] Yeong, W., Howes, T., and S. Kille, "Lightweight Directory Access Protocol," *RFC 1777*, March 1995.

[6] S. R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *Summer 1986 Usenix Conference Proceedings*.

[7] Chuck McManis, "Naming Systems: A Replacement for NIS," September, 1991, *Sun UK User Group Conference Proceedings*.

[8] Jan-Simon Pendry, "Amd – An Automounter," *Technical Report*, Department of Computing, Imperial College, London, England, 1989.

The Advancement of NFS Benchmarking: SFS 2.0

David Robinson – Sun Microsystems, Inc.

ABSTRACT

With the release of the Standard Performance Evaluation Corporation's (SPEC) System File Server (SFS) Release 1.0 benchmark suite in April 1993, the characterization of NFS server performance grew from a small and not widely accepted set of benchmarks, to a single industry standard benchmark. This paper will provide a historical look at the success of SFS 1.0[1] and how it has driven server capabilities, describe SFS 1.0's shortcomings, and detail the design and rationale behind the development of SFS 2.0.

Terminology

The SFS benchmark suite has been referred to by a number of different names that has lead to confusion. For this paper, the term SFS shall refer to the suite and in particular the framework used to drive the workloads, SFS 1.0 or SFS 2.0. Each suite is composed of one or more workloads which reflect the version of protocol used. SFS 1.0 contains the 097.LADDIS workload and SFS 2.0 contains the 162.nfsv2 and 163.nfsv3 workload. For this paper, the 097.LADDIS workload will be referred to as LADDIS, and 162.nfsv2 and 163.nfsv3 as V2 and V3 respectively.

SFS 1.0

Background

SFS is a synthetic benchmark used to measure the throughput and response time of an NFS server over a variety of load levels. The benchmark uses multiple physical clients, called load generators, each containing multiple load generating processes. Each load generating process is designed to represent multiple real world clients. A load generating process contains its own NFS and RPC protocol stacks, eliminating any effects due to different NFS client implementations, making it possible to use a variety of different hardware and operating system platforms and still have comparable results. The load generators send a controlled stream of time-stamped NFS requests to the server and measure the precise response times producing a detailed report of each type of operation, the overall throughput and average response time. The server is treated as a black box and the benchmark relies on no services on the server beyond the standard NFS protocol.

The SFS workload does not model any specific user application or any specific environment, but is designed to present the server with a series of requests to simulate the aggregation of many different applications and clients. The workload was designed through a series of studies of actual NFS traffic to servers used in a wide variety of environments.

The basic framework of SFS 2.0 has not significantly changed from the original SFS 1.0 release described in detail in the 1993 Usenix paper by Whittle and Keith [Whittle93]. The primary focus of SFS 2.0 was to update the accuracy of the workload and this paper will focus on those changes.

Historical Perspective

The SPEC SFS 1.0 benchmark suite and its sole component, the 097.LADDIS workload, has had a significant impact of the NFS server market since its introduction in April 1993. Prior to the release, evaluation and specification of NFS servers were done based on a number of small benchmarks that did not adequately represent real world NFS servers. Some of the problems included results that were greatly influenced by the effects of the client operating system, the lack of a defensible workload, and no common agreed upon standards for testing and reporting of results. The creation of the informal industry wide LADDIS group to address the technical issues and the subsequent incorporation with the SPEC standards body resulted in a benchmark that was accepted by the industry as fair and vendor neutral. The conventional wisdom that if you measure it, it will improve, is validated by the publication of results. Figure 1 shows a thirty fold increase in throughput over the five years results were published. This dramatic increase greatly exceeds the increase in processor performance over the same period. Using the often quoted figure of integer performance increasing every 12 to 18 months, the actual growth of LADDIS results exceeds this by a factor of 2 to 4. This growth can be attributed primarily to the enhancement of system software in efficiency and scalability on multiprocessor systems.

Another measure for the success of SFS has been the demand for results. In addition to the purpose built NFS servers, most vendors of general purpose servers now include SFS as one of the standard metrics marketed at their initial product announcements.

[1]SPEC SFS 1.1 was a bug fix release in 1994 which made no measurable changes to the workload. Both will be referred to as SFS 1.0 in this paper.

Customers are also including the SFS metrics in their minimum specification for both NFS and general purpose server requests for proposals.

Although SFS was designed primarily to produce competitive benchmarking results, the designed in flexibility and tuneability of a wide variety of parameters has enabled it to be used in the sizing of servers and creating application specific workloads. This capability has allowed both customers to evaluate their specific environments and server vendors to tune their systems for a multitude of applications.

Deficiencies

While SFS 1.0 has been extremely popular and a success in the industry, a number of deficiencies were known when it was released or subsequently discovered during its lifetime.

Operations Mix

The basis of the default percentages of each NFS operation (or operations mix) in the LADDIS workload was an unpublished 1986 study of the Sun Microsystems engineering network and also validated by a survey of customer nfsstat data from software development and other similar technical computing environments. The clients in the study were primarily a homogeneous set of workstations which may not reflect the differences in client implementations. Many of the workstations in the study were diskless and had relatively small amounts of physical memory resulting in small caches with high paging and swapping rates over NFS. Very few modern workstations are diskless and now contain large physical memories, resulting in large caches that reduce the need to swap over the network. Casual observation of current server loads appeared to indicate that the LADDIS operations mix was too heavy on I/O operations.

Version 3 and TCP

Version 3 of the NFS protocol was finalized shortly after the initial release of SFS 1.0. It provided some compelling new features, including better file system semantics and performance, resulting in it being chosen as the default protocol version on most major NFS vendor platforms. The benchmark, which measured only NFS version 2, needed to be updated to reflect what real customers were running in their environment.

Simultaneous with the release of NFS version 3, most vendors also introduced TCP as an available transport. While NFS was designed to be transport independent, there is some performance impact when compared to UDP due to the added complexity of managing a reliable transport compared to unreliable transport. With many vendors also making TCP the default transport, there was an increasing demand to characterize that impact.

File Size

The size of files created in SFS 1.0 is uniformly 136 kilobytes (KB) in length. The original LADDIS paper called this "unrealistic" but stated that it would

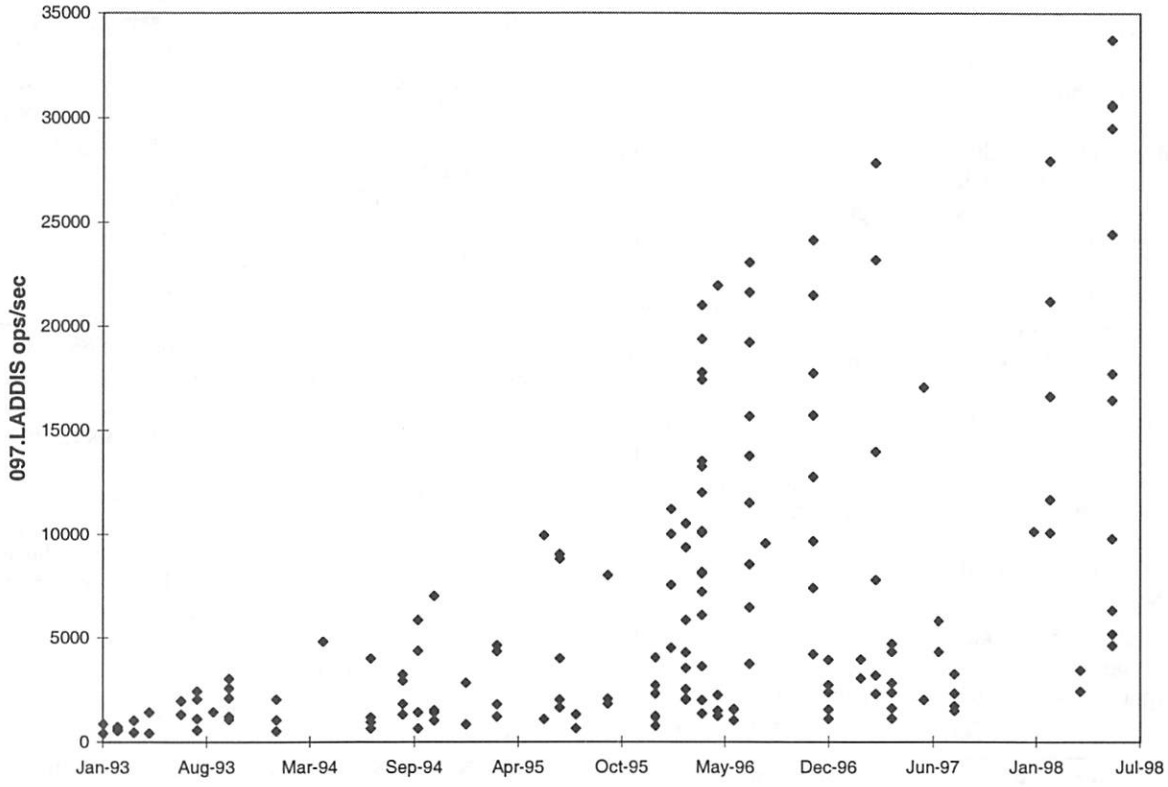


Figure 1: SPEC SFS 1.0 results published 1993-1998

have little impact on the performance model. With the appearance of servers that were specifically optimized for this size of file, the assertion was proven false and needed to be addressed.

Latency Metric

Even though the SFS 1.0 reporting pages include the response time at the peak throughput, the benchmark does not contain a metric to allow a fair comparison of the overall responsiveness of servers. The primary competition between vendors at the time of the release of SFS 1.0 was capacity, as no server was large enough to handle even a moderate sized work group. As server capacity (i.e., aggregate throughput as well as large storage components) has grown to be able to serve very large work groups, if not entire enterprises, a secondary market arose competing for small to medium size servers, using low response times as the key differentiator. The lack of a response time metric in SFS has limited customers ability to make fair comparisons in this new market.

Benchmark Portability

Great effort was taken to insure that SFS 1.0 could be run on a wide variety of client operating systems. While the source code was reasonably portable, it still contained a number of dependencies on the client's native RPC implementation and used parts of the client's native NFS implementation to initialize the file set. With the creation of 64-bit operating systems this reliance on the client NFS and RPC became problematic. Better portability was essential to the continued success of SFS.

Design of SFS 2.0

The design and motivation for SFS 2.0 was driven by two primary factors: creating a workload that more accurately reflects real world servers and updating it to support the recently released NFS version 3. The philosophy of SPEC is to provide standard benchmarks that reflect real world usage [SPEC], unfortunately every synthetic benchmark is a compromise. SFS 1.0 has a number of known deficiencies that were intentionally not addressed before the benchmark was released, and others that were discovered through use. By addressing some of the more significant compromises, customers will have an increased confidence in the reported results.

The changes required to update SFS 1.0 were primarily in the workload generated by the benchmark and the only minor changes were made to the framework and test harness. This is a credit of the work of the original designers.

NFS V2 Operations Mix

The LADDIS operations mix is roughly half file name and attribute operations, one third I/O operations, and the remaining one-sixth spread among the other operations. Throughout the years that SFS 1.0 was being used, the configuration of clients and the implementation of operating systems has steadily evolved. In the 1987 study, many clients were diskless, which is now relatively rare. Diskless clients must page in applications and swap over the network, resulting in a substantial number of I/O operations. This was reflected in the mix by 37% of the operations being reads or writes. Additionally, the average amount of memory configured on a client has risen from 16 or 32 megabytes (MB) of memory, to 128 MB or more. The additional memory allows clients to aggressively cache more data on the client, further reducing I/O operations.

Based on the understanding of the change in the nature of clients and some informal validation based on actual observations, it was believed that the SFS 1.0 workload was too heavy with respect to I/O operations. A formal study was conducted to collect the actual operations mixes of over 750 Auspex servers running NFS V2.

A cluster analysis of the data collected was performed using the SAS statistical tools to determine if there were any significant common mixes present. Only two dominant clusters emerged, the primary cluster (75%) contained an operations mix that was very similar to the existing LADDIS workload, but as predicted, there was a significant reduction in the number of I/O operations. It contained approximately half the read operations and one third the write operations. The secondary cluster (15%) was very I/O intensive and contained a significantly higher percentage of read operations and a slightly higher percentage of write operations than the existing workload. The remaining 10% of the data was spread across a wide variety of mixes, none of which were statistically interesting.

Operation	Cluster 1	Cluster2
null	1%	0%
getattr	25	15
setattr	1	1
lookup	40	23
readlink	7	2
read	11	32
write	5	17
create	2	1
remove	1	0
readdir	7	5
fsstat	1	2

Table 1: NFS V2 Mix Cluster Analysis.

The goal of SPEC benchmarks is to provide customers with a workload that represents a common real world environment. The cluster analysis clearly showed one dominant operations mix, but from the raw data it was not known if this represented one or many types of environments. Each of the servers were further categorized by the type of environment (e.g., software development, office, CAD) and a statistical

correlation was run against the clusters. The primary cluster surprisingly showed no significant correlation to any specific type of environment. The secondary cluster showed a correlation to the computer aided design (CAD) environments. CAD is known to be very I/O intensive as it reads and writes large design data sets, confirming the correlation.

The default V2 workload was chosen to match the primary cluster based on it representing the vast majority of servers studied and the lack of correlation to any one environment. A second workload matching the secondary cluster was considered but ultimately rejected. Although this cluster showed a strong correlation to a CAD environment, the workload typically consists of large sequential reads and writes which can be more easily simulated by a simple copy benchmark.

Operation	097.laddis	162.nfsv2	163.nfsv3
getattr	13%	26%	11%
setattr	1	1	1
lookup	34	36	27
readlink	8	7	7
read	22	14	18
write	15	7	9
create	2	1	1
remove	1	1	1
readdir	3	6	2
fsstat	1	1	1
access	-	-	7
commit	-	-	5
fsinfo	-	-	1
readdirplus	-	-	9

Table 2: SFS Operation Mix

NFS V3 Operations Mix

At the time of the development of SFS 2.0, the deployment of NFS V3 within the industry was not yet widespread. Major operating system vendors had not yet, or had only just recently, released support for V3. The detailed statistical analysis used to derived the V2 workload was not possible due to the lack of a significant installed base. Because support of V3 was a key requirement for the release of SFS 2.0, a reasonable approximation of a V3 workload was required.

The operations mix presented to the server represents the aggregation of the file system workload generated by applications running on clients and then converted into NFS operations by the client operating system. In moving from V2 to V3, the application workload remains constant, so a transformation from V2 to V3 is possible if a typical client implementation is known. Using the data from a published comparison of a V2 and V3 client running the Andrew benchmark [Pawlowski94] the V3 operations mix was partially derived. The resulting mix was confirmed through a survey of servers within the Sun engineering network that served both V2 and V3 clients.

The V3 workload appears to be more data than attribute intensive when compared to the V2 workload. This is a result of the the changes in the underlying protocol, in particular, most V3 operations return the attributes of a file, thus reducing the number of getattr operations requested by the client. Through the addition of the readdirplus operation that returns the attributes with each directory entry, the number of lookup operations is also reduced. The result of the heavier V3 operations is a numerically smaller throughput value even though the server can handle more clients.

The commit operation had to be treated specially, as the number of commits is derived from the number of write operations and not measured experimentally. A commit is generated by a client to request that a server flush to stable storage any data written by previous asynchronous write operations. The most common scenario resulting in a commit operation is the closing of a file, when the client wants to free up the resources it may have been caching.

Within SFS, the write requests are typically the same size as the file being written to simulate the common practice of writing an entire file. These requests are then broken down into one or more write operations of the transfer size (8192 bytes) length. For requests less than or equal to the transfer size, only one write operation is generated and it is done synchronously. For all other requests, the commit operation is generated after the final write to simulate the close. With just over half of all write requests generating more than one write operation, 4% of the total workload are commit operations.

Length (KB)	Read	Write
1-7	0%	49%
8-15	85	36
16-23	8	8
32-39	4	4
64-71	2	2
128-135	1	1

Table 3: SFS 2.0 Percentage of File I/O Operations.

File Set

The fundamental properties of the SFS 2.0 file set has not changed from SFS 1.0. A large static set of test files is created consisting primarily of regular data files and directories and a small fixed number of symbolic links. No files are shared between load generating processes which reflects the absence of sharing in real world environments.

The number of data files and directories scales with the requested load like SFS 1.0, but at a ratio of 10 MB of data for each NFS operation/second (op/sec) instead of 5 MB per op/sec. The increase reflects the actual growth in disk capacity and that disk usage closely tracks the capacity growth. A larger file set causes servers to further stress their caching algorithms and disk seek times. The intention behind this was to emulate real world server conditions.

Working Set Size

From the large file set created, a smaller working set is chosen for actual operations. In SFS 1.0 the working set size was 20% of file set size or 1 MB per op/sec. With the doubling of the file set size in SFS 2.0, the working set was cut in half to 10% to maintain the same working set size. Although the amount of disk storage grows at a rapid rate, the amount of that storage actually being accessed grows at a much slower rate. In the IBM study, discussed below, it was found that only 12% of the storage was accessed within the last five days. A 10% working set size may still be too large. Further research in this area is needed.

Variable Sized Files

The size of files created by SFS 1.0 was uniformly fixed at 136 KB. The fixed size allows for a simpler implementation and a less complex selection algorithm to determine which I/O file to use. The size of 136 KB was chosen to be large enough to handle a variety of starting file offsets for read and write operations, while ensuring that the file system on the server allocates and references the first-level indirect file index blocks. At the time, it was believed that this compromise would have little impact on performance. In addition, other studies performed showed that most write operations are sequential, resulting in a high percentage (70%) of operations appending to a file. Unfortunately the combination of fixed size files and the high append ratio allowed servers to significantly increase their results by optimizing the handling of the first-level indirect file index blocks. Because most files in the real world are small and do not utilize indirect blocks, the increase in the benchmarking result from this optimization did not translate into a real world increase in performance. To solve this problem the SFS 2.0 file set uses data files that are of varying size.

A study was performed to determine a typical distribution of file sizes. The IBM engineering network containing over 6 million files and directories with a total storage of over 210 billion bytes was used as the basis of the study. For each file, the size, the number of fragments, the last access time, the last modification time, the length of the name, and the distribution of characters in the name, were recorded.

A simple statistical analysis showed that the median file size was just under 2048 bytes with 76.7% of the files under 8192 bytes. However, these files represented only 4.8% of the space consumed. Only 0.5% of the files were over one megabyte, yet represented 48.8% of the space consumed. The data very closely reflects a study performed three years earlier [Irlam94] with a slight trend towards more files with larger sizes.

When the IBM study's distribution is compared to the SFS 1.0 file distribution, I/O performed on 136 KB files only represents less than 4% of the files, confirming the concerns about using fixed length files. To address this problem SFS 2.0 creates a file set that is composed of files varying in size from 1 KB to 1024 KB, a range that reflects the experimental data. The read and write size distribution (Table 3) has not changed from SFS 1.0, as a result the large 1024 KB files are never accessed. Because these large files are

File size (max. bytes)	Number Files	Percent Files	Percent Space
0K	101264	1.69%	0.00%
0.5K	1102362	18.36	0.49
1K	746062	12.43	0.33
2K	1252118	20.86	1.11
4K	776916	12.94	1.18
8K	622675	10.37	1.71
16K	461333	7.68	2.43
32K	317392	5.29	3.25
64K	226765	3.78	4.62
128K	159661	2.66	6.57
256K	104109	1.73	8.19
512K	77396	1.29	12.87
1024K	26553	0.44	8.47
2048K	14396	0.24	9.05
4096K	8659	0.14	10.75
8192K	3410	0.06	8.46
16384K	1420	0.02	7.03
32768K	616	0.01	5.93
> 65536K	231	0.00	7.56

Table 4: IBM File Size Distribution.

such a low percentage of the total, no change in performance is expected.

While the distribution of file sizes is correct, the average file size is only 27 KB instead of the 37 KB average found in the IBM experiment. This was not noticed until after the release of the benchmark. A better distribution would be to replace the one percent of 1024 KB files with one percent 2048 KB files.

Size (Kbytes)	Percentage
1	33%
2	21
4	13
8	10
16	8
32	5
64	4
128	3
256	2
1024	1

Table 5: SFS 2.0 File Size Distribution.

File Selection

With all files in SFS 1.0 being fixed in size, the selection of a file for an I/O operation within the working set is done randomly with a uniform distribution. A serious problem resulting from this approach was the common occurrence of a small read request starting from the beginning or middle of a 136 KB file. Many servers and disk subsystems are tuned through read ahead algorithms for the common case of an application reading a file from start to end. The fixed size selection algorithm resulted in some servers demonstrating worse benchmark than real world performance.

The random selection of I/O files will not work with the new variable file size distribution, as some files may not be large enough to satisfy the request. Both a first fit and a best fit algorithm would allow all requests to be satisfied, but a best fit was chosen because it would also maximize the number of whole file reads and writes.

The variable file size distribution results in 67% of all files being less than the 8 KB transfer size. However, most modern NFS clients use an I/O subsystem that is integrated with the virtual memory subsystem, resulting in all I/O being rounded up to page size requests. To simulate this common style of client, SFS does not request the actual size of a small file, but instead issues a full 8 KB read. When a request for a read of less than 8 KB is made, any file less than 8 KB is chosen on a first fit basis. This greatly simplifies the selection algorithm while not materially changing the load on the server.

Append ratio

Research at the time of the original SFS 1.0 release indicated that the ratio of writes appending to files was as high as 90-95% in some environments [Hartman92]. A simple explanation of this is the fact

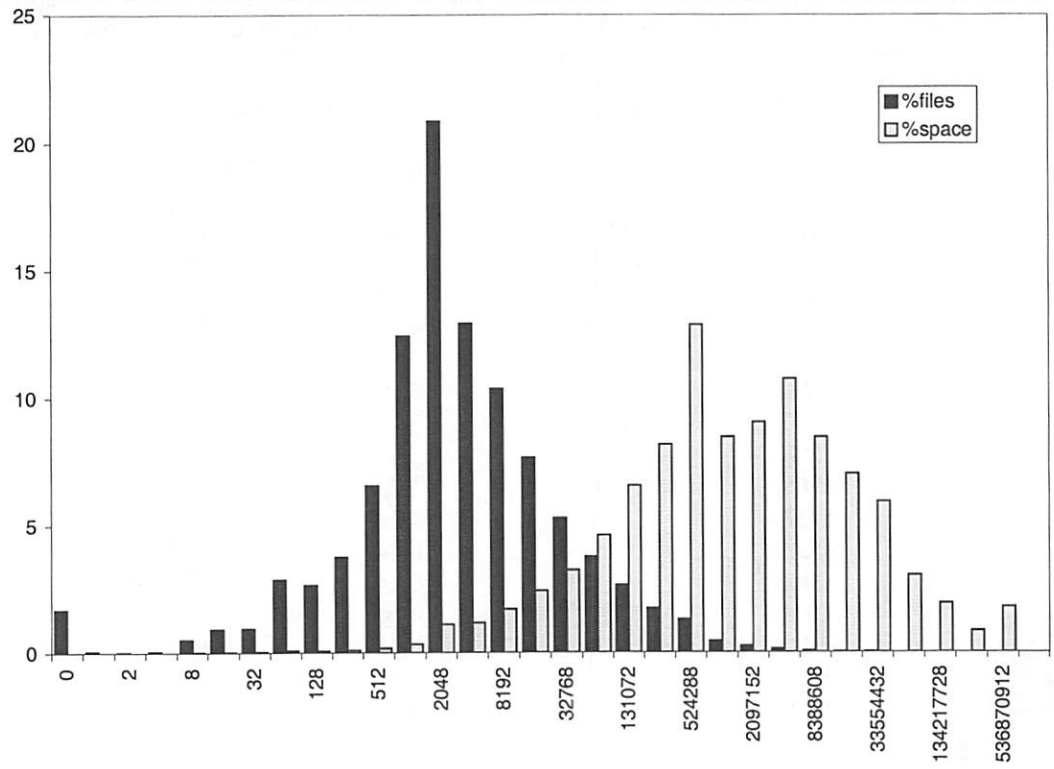


Figure 2: IBM distribution of files by number and space.

that most applications write a file from start to end. Unfortunately it was found that the benchmark became unstable due to uncontrolled growth of the file set when append ratios higher than 70% where chosen. SFS 2.0 maintains the same explicit append ratio but the actual append ratio is higher through the combination of variable sized files, a best fit selection algorithm, and most requests resulting in multiple sequential operations to the server. Although this better simulates whole file writes, the append ratio is probably still not high enough.

Multiple Directories

Each client in SFS 1.0 runs one or more load generating processes, each with its own directory of files to operate on. The directory is large and flat, containing all of the files and empty directories. While easy to implement in the benchmark, the main directory became unrealistically large under heavy loads causing an over-emphasis on the server's ability to perform directory operations. SFS 2.0 solves this problem by creating a new subdirectory for every 30 files created in the working set.

```
int clnt_poll(CLIENT *cl,
              uint32_t usecs);

bool_t clnt_getreply(CLIENT *cl,
   xdrproc_t xres, void *resp,
   int cnt,
uint32_t *xids, uint32_t *xid,
          struct timeval *tv);
```

Table 6: New client interfaces.

The IBM study showed that the mean number of directory entries was 10 but the median was only 3, significantly less than the 30 entries created by SFS. To accurately implement this distribution would have been fairly complex. The performance impact of this compromise is considered to be low. The operations most affected by directory size are lookup, readdir, and readdirplus. Virtually all servers implement some form of directory name lookup cache which minimizes the effects of directory size on lookup operations. Conversely, using a small number of entries in a directory would help the performance of readdir operations but would not reflect some significant environments. By choosing a fixed size of 30 entries, lookup operations are not greatly effected by directory size and there are enough entries to adequately test readdir performance.

The names of the files created in the file set were changed to reflect the type of the file. This was done to assist in trouble shooting benchmark problems as well as to invalidate any potential server optimizations based on the SFS 1.0 names. The average length of a file name was set to 13 to reflect the results of the IBM study which showed that over 50% of the files had a name length between 8 and 12 characters. The IBM study also provided the average occurrence of a particular character within a file name. Although no attempt was made to set the selection of characters in a file name to match this distribution, the most common character, the period, was added to the names. The current simple fixed naming scheme is still vulnerable to servers optimizing caches for the benchmark based on the names of files.

TCP

The NFS protocol is designed to be transport independent, however until the early 1990 most implementations chose to only support the UDP transport. Concurrent with the release of V3 implementations, most vendors also introduced support of both V2 and V3 over the TCP transport and many made TCP the default. To provide customers with the ability to compare servers running in these environments, support for TCP was added to SFS 2.0.

Connection Management

There is no formal specification of how many TCP connections should be made between a client and a server. Some clients choose to have just one connection, others may have one per mount point, while some user level clients may have one per file. SFS is designed to present the server with the appearance of many clients but only using a small number of actual load generators, each generating a pseudo random set of operations on a set of files. This is very effective at hiding the number of real clients when using a protocol like UDP that has a single end point. However, with a connection oriented transport like TCP, the choice of how to model connections is quite important.

Although there are multiple choices on how a client should connect to a server, the majority of the clients have only one connection to each server. This is fortunate, because the aggregation of the operations of many client processes over multiple mount points can still be reasonably modeled using SFS' pseudo random operation selection. If one connection per mount point, or worse one per file, were common, it would have required SFS to introduce temporal effects into the operation selection, a formidable task.

To simulate multiple connections, the existing SFS 1.0 framework that supports multiple load generating processes on each load generator was used. Each load generating process creates one connection to the server. While the number of load generating processes used on tests of very large systems is likely to be considerably less than would occur in a real world environment, the effects were not considered to be significant in the results.

Implementing TCP

The addition of TCP to the SFS 2.0 code base proved to be one of the more difficult challenges. A primary feature of SFS is the ability to simulate client asynchronous read-aheads and write-behinds, otherwise known as BIOD emulation. Because of the request/response nature of an RPC protocol, a client

waiting for the result of an I/O operation to return before issuing the next request, would suffer from poor sequential performance. To minimize this problem, most clients break large I/O requests into multiple concurrent requests and perform read-ahead or write-behind.

To minimize the effects of the client operating system and increase portability, SFS provides a full NFS and RPC user level client. Unfortunately, for portability reasons the client is single-threaded, making BIOD emulation a challenge. To further compound the problem, no freely available RPC library provided the capability to send RPCs asynchronously and gather replies. Ironically the existing libraries contain a minimal attempt at sending asynchronous RPCs, but fail to provide a reply gathering mechanism.

BIOD emulation was built using the freely available Sun RPCSRC 4.0. The standard *clnt_call* function was split into three pieces, the synchronous case and two new asynchronous functions *clnt_getreply* and *clnt_poll*. The standard *clnt_call* semantics were extended to place the RPC's transaction id in the result buffer if a timeout of zero is specified, allowing the caller to track multiple requests. The client can then call the new *clnt_poll* with the list of transaction ids to wait on a reply. When a reply is available *clnt_getreply* is called to process one request. The client provides as arguments an array of transaction ids being waited upon and if the *clnt_getreply* is successful, it returns the transaction id of the reply that was processed. The client can then repeat the process until all outstanding requests are processed.

Transport Specific Metrics

The choice of using TCP or UDP as the transport layer will have an impact on the performance of NFS. It is generally believed that UDP as a lighter weight protocol will offer better performance on a local network where packet loss is (nowadays) extremely rare, while TCP, with its reliable nature, will offer better performance on a wide area network, where packet loss is not uncommon.

The interesting question was whether to create an additional transport specific metric for both V2 and V3 or not. Because there is a measurable difference between transports, some considered it unfair to compare a TCP result against a UDP result. However, the workload presented to the server is independent of the transport. If the performance effects caused by choice of transport layer are compared to the performance effects caused by choice of network media, many similarities exist. An FDDI ring will have lower latencies than a 10BaseT Ethernet and a network adapter that does on board protocol processing will have higher throughput than one without. In these cases no distinction in the metric is made and therefore the choice of transport is also not reflected in the metric. The transport is clearly presented in the full results report allowing an informed comparison to be made.

Overall Response Time Metric

The official abbreviated form of an SFS 1.0 results report is the throughput in operations per second at a peak response time in milliseconds. While the primary throughput metric is easily understood and comparable between systems, the response time is poorly understood and is occasionally misused in the marketing or interpretation of results. The response time reported is simply the response time at the maximum throughput with the only condition that it can not exceed 50 milliseconds. To compare the peak response time of two results only shows how the system degrades under maximum load and is no reflection on how the system response during typical loads experienced by most customers.

As the capacity of most NFS servers has grown dramatically, the aggregate throughput of the server has become a secondary consideration for many environments. The responsiveness of the server becomes more important as it contributes to the throughput of sequential operations as well as the subjective "feel" experienced by end-users sitting at NFS clients.

In comparing two SFS results, the shape of the curve is an indication of how the server responds over the entire load range. A curve that is flat and close to the x-axis indicates a server that is consistently fast over a wide range of loads. The challenge was to create a meaningful single metric that captured the quality of the graph. The area under the curve divided by the peak throughput was chosen as the new SFS 2.0 metric called the Overall Response Time, calculated by using the Riemann sum:

$$O.R.T. = \frac{\sum_{0}^{n} \left[\frac{R_i + R_{i-1}}{2} * (T_i - T_{i-1}) \right]}{T_n}$$

Run Rules

A critical component of any SPEC benchmark are the run and disclosure rules which allow customers to make fair comparisons between competing vendors. To the credit of the original SFS 1.0 benchmark, only a few significant changes were required for SFS 2.0.

Response Time

The throughput of a system is the primary metric for comparison between two competing vendors, but at some point additional throughput is overwhelmed by the degradation in response time. The maximum reportable response time was set in SFS 1.0 to the some what arbitrary value of 50 milliseconds. The choice was based primarily on the human sensitivity of response time. Following the decrease in both network and disk latency, the maximum reportable response time was decreased to 40 milliseconds in SFS 2.0.

Warmup and Runtime

SFS load generation is split into two phases, warmup and runtime. The warmup phase is required to allow the benchmark to make initial requests and adjust the

inter-request sleep interval to achieve the requested load level. As vendors gained experience with SFS 1.0, it was observed that in some configurations the benchmark would not converge on a steady request rate within the one minute warmup time resulting in an undesirable variance in reported results. For SFS 2.0, the warmup time was increased to five minutes to ensure stability of the request rate. Additionally it was observed that during the last five minutes of the ten minute runtime, the request rate was extremely stable and the results were unaffected by decreasing the runtime to five minutes. The increase in the warmup time may cause large servers to better cache the working set, it is hoped that the decrease in

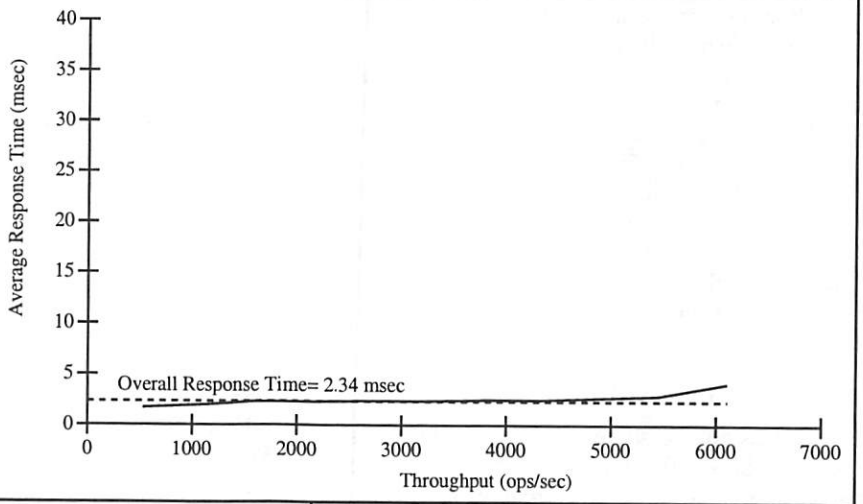
SPECsfs97 Result

Blaster Server Inc.
Ultra SuperBlaster 2001

SPECsfs97.v2 =
6104 Ops/Sec (Overall Response Time = 2.34 msec)

Tested:	Sep 1999	By:	Blaster Server Inc.	License #:	666	Hardware Avail:	Sep 1999	Software Avail:	Sep 1999

Throughput (ops/sec)	Average Response (msec)
530	1.7
1171	2.0
1602	2.3
2150	2.3
2700	2.4
3240	2.4
3780	2.5
4320	2.5
4860	2.7
5450	2.9
6104	**4.1**



CPU, Memory and Power

Model Name:	Ultra SuperBlaster 2001
Processor:	4 GHz SB Mach 3
# of Processors:	2
Primary Cache:	16KBI+512KBD on chip
Secondary Cache:	1GB(I+D) off chip
Other Cache:	N/A
UPS:	N/A
Other Hardware:	N/A
Memory Size:	20 GB
NVRAM Size:	512 MB
NVRAM Type:	Static
NVRAM Description:	Nuclear Powered

Server Software

OS Name and Version:	CPM 3.0
Other Software:	N/A
File System:	FAT16
NFS version:	2

Network Subsystem

Network Type:	1000Mbit GigabitEthernet
Network Controller Desc.:	GigabitEthernet PCI adaptor 2.0
Number Networks:	1 (N1)
Number Network Controllers:	1
Protocol Type:	UDP
Switch Type:	1 Blaster 3 switch
Bridge Type:	N/A
Hub Type:	N/A
Other Network Hardware:	N/A

Server Tuning

Buffer Cache Size:	Dynamic
# NFS Processes:	945
Fileset Size:	57.3 GB

Disk Subsystem

Number Disk Controllers:	3
Number of Disks:	136
Number of Filesystems:	132 (F1-F132)
File System Creation Ops:	default
File System Config:	default
Disk Controller:	On-board SCSI-III
# of Controller Type:	1
Number of Disks:	4
Disk Type:	9GB 7200rpm SCSI-III
File Systems on Disks:	OS, swap
Special Config Notes:	
Disk Controller:	On-board SCSI-III
# of Controller Type:	11
Number of Disks:	66
Disk Type:	9GB 7200rpm SCSI-III
File Systems on Disks:	F1-F66
Special Config Notes:	
Disk Controller:	On-board SCSI-III
# of Controller Type:	11
Number of Disks:	66
Disk Type:	9GB 7200rpm SCSI-III
File Systems on Disks:	F67-F132
Special Config Notes:	

For More Information contact

SPEC
10754 Ambassador Drive, Suite 201
Manassas, VA 20109

(703) 331-0180
info@spec.org
http://www.spec.org/

Page 1 of 2

Figure 4: SFS 2.0 Reporting Page

the runtime will lessen any opportunity to exploit the cache.

Clients per Network

The SFS 1.0 run rules require that there be at least two clients per network used in the test configuration. The initial intent was to ensure that clients would realistically experience collisions on ethernet networks. A few factors have emerged to make this requirement unnecessary. The wide spread shift from fat yellow ethernet cable and thin-net to switching twisted pair intelligent hubs have greatly reduced the chances that a client on a heavily loaded ethernet will experience a significant number of collisions. Even on a network with collisions, the effects of the physical layer arbitration does not change the maximum capacity of a server. When a client backs off from a collision, it increases the response time for a given operation but does not decrease the capacity of the server, as the back off is handled in hardware without intervention of the main processor. The collision does decrease the available network bandwidth, but with the addition of another network interface the same server can handle the same capacity whether the network is switched or not. With the goal of SFS to measure server capacity, the mandating of network topology has proven to be unnecessary.

Future Work

SFS 2.0 was able to solve a number of the flaws in the original SFS 1.0 benchmark and update the workload to more closely reflect real world usage. However there are still a number of significant areas for future work. Most important is another study of V3 servers, similar to what was performed for V2 servers, to validate the operations mix. In the time since the release of the benchmark, it appears that the mix of directory reads may need to be adjusted. The algorithms used by clients to determine when to issue a light weight readdir request, versus a readdirplus request, is still evolving and moving towards minimizing the number of readdirplus operations.

The SFS framework is designed to provide the correct operations mix and I/O sizes based on studies that represent the cumulative history of the server. The random selection does not take into account the temporal nature of file access. The next file accessed is more likely to be the same file, or a file in the same directory, than a file randomly selected from the working set. Two possible solutions would be to study the sequences of requests made to servers to determine any significant patterns, or to switch from a random selection with a Poisson distribution to a Markoff distribution.

SFS began as a benchmark that used the client system calls to generate network traffic. It has become client independent, generating network traffic directly. The reduction in client sensitivity allows for very fair comparisons of two servers using the same protocol. But customers are asking for comparisons of the same server using different protocols. To accomplish this a protocol independent benchmark that either calls the client system calls directly or drives real or pseudo applications. The challenge of such a benchmark is to determine what a representative workload is and to manage client side effects so the server is being fairly tested. The Andrew benchmark was an early attempt at a protocol independent benchmark.

Acknowledgments

The author would like to acknowledge the members of the SFS steering committee who spent many hours collecting data, analyzing it, testing the code, and most importantly, arguing about it. In particular, Kamesh Gargeya, Sherry Hoffman, Joan Lawler, Tony Lukes, Brian Pawlowski, Judy Piantedosi, Spencer Shepler, Pete Smoot, and Andy Watson.

This paper is dedicated to the memory of Terry Flynn who provided most of the statistical analysis which is the basis of SFS 2.0.

Author Information

David Robinson is a Senior Staff Engineer in the Solaris Networking Technology Group at Sun Microsystems, Inc. He has been the primary author of the SFS 2.0 benchmark suite, and founder of the SPEC SFS steering committee. He has been active in the NFS development community for since 1986. Prior to Sun, he worked at the Jet Propulsion Laboratory where he developed one of the first NFS servers for VMS to allow workstations to access large satellite images. Reach him via U.S. Mail at Sun Microsystems, Inc., MS MPK17-201, 901 San Antonio Rd, Palo Alto, CA 94303.

References

[Baker91] Baker, Mary G., "Measurement of a Distributed File System," *Proceedings of the 13th Symposium on Operating System Principles,* pp. 198-212, October 1991.

[Callaghan95] Callaghan, Brent, et al., "RFC1813, NFS Version 3 Protocol Specification," June 1995.

[Hartman92] Hartman, John, "File Append vs. Overwrite in a Sprite Cluster," Sprite Project, University of California at Berkeley, Presentation to the LADDIS Group, Jan 21, 1992.

[Irlam94] Irlam, Gordan, http://www.base.com/gordoni/ufs93.html, *An informal study of Unix file sizes from data gather via Usenet*, Sept, 1994.

[Keith90] Keith, Bruce, "Perspectives on NFS File Server Performance Characterization," *Proceedings of the Summer 1990 USENIX Conference*, pp. 267-277, June, 1990.

[Pawlowski94] "NFS Version 3 Design and Implementation," *Proceedings of the Summer 1994 USENIX Conference*, June, 1994.

[Sandberg85] Sandberg, Russell, et al., "Design and Implementation of the Sun Network File System," *Proceedings of the Summer 1985 USENIX Conference*, pp. 119-130, June, 1985.

[SPEC] Standard Performance Evaluation Corporation (SPEC) web site, http://www.spec.org/ .

[Sun89] Sun Microsystems, "RFC1094, NFS," March, 1989.

[Watson92] Watson, Andy, et al., "LADDIS: Multi-Vendor and Vendor-Neutral SPEC NFS Benchmark," *Proceedings of the LISA VI Conference*, pp. 17-32, October, 1992.

[Whittle93] Whittle, Mark, et al., "LADDIS: The Next Generation In NFS File Server Benchmarking," *Proceedings of the 1993 USENIX Conference, 1993*.

Moving Large Filesystems On-Line, Including Exiting HSM Filesystems

Vincent Cordrey, Doug Freyburger, Jordan Schwartz, and Liza Weissler – Collective Technologies

ABSTRACT

Since the advent of Logical Volume Managers [LVM], larger individual disk drives, and high uptime expectations, it is no longer possible at some sites to schedule downtime windows long enough to move some very large or very critical filesystems to new hardware. Hierarchical Storage Management [HSM] systems share this problem. While at some sites, users continue to enjoy the functionality of HSM based on their specific usage patterns, sites whose usage patterns do not match HSM's strengths have a more acute case of the same problem when moving their data to new hardware. This paper presents a unique approach to moving filesystems that *permits a system to remain on-line and accessible.* New terminology is also introduced to assist discussion: **Forward Relocation**, **Reverse Relocation**, and **Hybrid Relocation** are defined and basic algorithms are presented. While it is true that the total throughput rate of a traditional dump and restore is higher, the methods presented here require nearly zero downtime.

The authors have used these techniques to relocate data on filesystems with many small files during the working day at several sites, as well as to exit HSM systems as part of standard technology refresh programs. Three case studies, where both types of data were relocated, are described in their basic detail as successful (and ongoing) implementations. The authors know of no prior works on this topic and hope to foster further discussion and refinement of the techniques.

Introduction

In plotting the rehosting of large filesystems long outage windows seem inevitable. With the ideas presented here, the authors hope to expand the rehosting debate by potentially reducing gaps in data availability traditionally associated with a relocation of data from one server to another or even within the same server. While not a perfect solution for all filesystem relocation projects, some installations may find the nearly continuous data availability attractive enough to consider implementing one of these techniques.

Background and Terms

Uptime Expectations

As the total user population increased, computing pervaded the corporate desktop and email is now a mission critical service. Thus, users have come to expect production uptime from fileservers. These users are not interested in computers for their own sake; rather, they use their systems as tools, so they are less tolerant of outages. The standard has gone from users who viewed their computers as sports cars that they expected to tinker with, to users who view their computers as telephones with screens. This leads to the expectation of a "dial tone" whenever they reach for their keyboards.

So Many Files, So Little Time ... For Now

Users accumulate very large numbers of files because they hoard their data and email messages for *years.* Nearly all methods for copying filesystems take much longer with large numbers of files than with fewer files taking up the same amount of total storage. The longer copy times can mean that it may take an entire shift to copy a large filesystem (even when a full weekend is prepended to the outage window) and users will not tolerate that much downtime.

Logical Volume Managers [LVM] aggregate many disks into very large filesystems. Also, some systems use automounters to supply home directories to their users from many filesystems. In either case, the users now demand high uptimes, and long scheduled outages are not acceptable at some sites. Thus, the ability to relocate filesystems **On-Line** during the day shift is important, especially at some sites that are short-handed in support personnel.

As faster filesystems and networks are designed, this trend will cease, but for the moment, the pendulum of changing technology has given rise to filesystems too large to relocate off-line during an acceptable outage window, and has created the need for **On-Line Relocation**.

Terminology

Early work with this subject revealed that, to prevent confusion, we needed to establish some fairly strict terminology. Even this paper required careful corrections of usage.

This subject deals with the movement of files, directories, and indeed, whole filesystems. Whether it

is a discussion of HSM, our new techniques, or traditional methods, data and files ebb and flow from one place to another. Thus, we have tried to choose language that permits the type of movement to be differentiated. We have adopted vendor specific terms, industry usage, and attempted to eliminate overloading.

Basic HSM Terms

medium the smallest discrete storage unit addressed as a whole; a tape volume, optical platter, physical disk, logical volume, etc.

migration – the movement of files to deeper levels within HSM Systems; implies the probable return of the file to its original location.

migrated – absent from the medium (level) being examined as a result of migration.

resident – opposite of migrated; object is present on the medium (level) being examined.

stage-in – to recall from a deeper level of HSM and make resident.

stage-out – to migrate to a deeper level of HSM and remove from source medium.

Paper Terms

full both source and destination filesystems or directories remain in read-write mode.

semi at least one of the source or destination filesystems or directories is not read-write.

relocation – movement of files from one filesystem or directory structure to another as different from migration; implies a single, permanent movement of the file.

scatter placed and retrieved from non-contiguous locations, different media, or different directories.

systematic – placed and retrieved from contiguous locations, the same medium, or the same directories.

HSM General Principles

Hierarchical Storage Management [HSM] refers to software that permits automatic migration of data from on-line storage, usually magnetic disk, to lower cost secondary and perhaps tertiary storage such as optical disk or tape. UNIX implementations of HSM came to the fore in the late 1980's. HSM helped answer the call for very large storage systems at a time when large capacity spinning disk servers for UNIX systems were expensive and not commonly available outside of the realm of super computers.

HSM systems work very well for some types of data, and many sites continue to enjoy excellent service from them. However, some early HSM adopters now find themselves with aging systems that are at or near the end of their useful life, plotting the relocation of files either to large fileservers, newer brands of HSM, or true archival systems.

HSM systems perform well with a small number of large files but are commonly used at installations where there are a large number of small files. HSM systems have a high storage processing overhead and are therefore inappropriate for small files.

HSM is philosophically different from true archival systems, but it is commonly abused as such. Multiple copies back to a baseline or some limitation on the number of copies is an "oops" recovery feature, not an archive, and an "undelete" feature is not explicitly supplied on many systems. A trash can recovery feature is also not an archive. More to the point, an HSM system is designed to keep accessible nominally one *most recent* copy of a file. Archiving involves keeping a specific copy of a file that represents the state of the file at a specific time. Additionally, an archiving system allows access to a large number of versions of a given file that represent specific versions accumulated over time. While this may sound like a revision control system, archiving can be used even with files that are too large for reasonable differencing, and archiving normally also involves storage on less expensive media that may be near-line or off-line. Good archive systems also index all of the versions of the files that have been archived – they keep track of multiple versions of the same file with the same name.

What Changed?

Several factors have changed the technologies so that large capacity fileservers are now common, and HSM systems are no longer the only choice for storage of large amounts of data:

- Spinning disk media is far less expensive and much larger single disk drives are now available.
- LVMs are now available that allow systems to aggregate many disks together into very large filesystems.
- There are now vendors that specialize in large capacity, high performance fileservers.
- On the user interface front, "drag and drop" GUIs encourage misuse as users copy entire directory trees wholesale from place to place without regard to where the data might be stored.
- The labor costs to maintain HSM systems are high: the systems are multilayered, complex, and require highly skilled labor to keep them running.
- An unanticipated aspect of HSM labor costs is the fact that both media volumes and databases require manual garbage collection. This requires a high degree of skill to accomplish, and without it, performance gradually declines.

Why Relocate in the First Place?

The simple motivation to *relocate* the data rather than *abandon* it is that you want to *keep* it. At sites running large non-HSM fileservers, a constant technology refresh program is required to stay current: additional, larger, faster disks are installed; different

filesystem software is added (such as journaled filesystems); and entire servers are replaced. In these cases, data on previous generations of hardware and filesystems must be relocated. This was previously done during a scheduled outage window. Given large amounts amount of data and the time it takes to relocate it, the ability to relocate data on-line during the working day has become extremely useful.

Contemporary with the changes in technology, several reasons to exit HSM systems have surfaced. HSM systems, by their very nature, do not provide real-time access to all of the data. With everything a click away, waiting is no longer acceptable. Diminishing HSM expertise makes the systems hard to maintain in an operational state – *everyone who knew how to deal with it left and you're stuck holding the bag.* Backups on some HSM systems can be slow to complete and add extra layers to a backup scheme, as a full backup now only represents data resident on spinning disk. The internal complexity of HSM systems can make them unstable; HSM systems have several failure modes. Some systems suffer from inter-component communications failures which lead to an interruption of service that may require a full system reboot to clear. Media failures plague some installations, while index databases and data files can be corrupted by filled filesystems on some others. Lastly, because the systems were designed in the late 1980s, some of them are not Y2K compliant.

Techniques

The technique used to relocate the data from a filesystem depends on three basic choices:

- File selection (**scatter gather** vs **systematic**)
- Data availability (**Full On-line** through **Off-line**)
- Relocation algorithm choice (**Forward**, **Reverse**, **Hybrid**, or **Just Plain Copy**)

File Selection

File selection algorithms largely do not matter when non-HSM systems are being relocated: the files are usually on the same medium, and there is a very small penalty for selecting them at random. With HSM systems, however, the algorithm used can have a significant impact on the time required to relocate the data. While this section discusses two basic file selection algorithms as they apply to HSM systems, there may be some cases when the concerns addressed here should be applied to non-HSM systems.

A Word About Scatter Gather Versus Systematic

HSM media volumes, be they tape or optical, are created as needed. This means that the mapping between files in an HSM medium and files in a directory appears to be random. Some HSM systems deliberately try to distribute the files onto a larger number of media to limit the impact of losing any one medium by spreading the migrated files across many media. So any given directory or tree will likely have files on many media volumes.

The definition of the Scatter Gather technique is to ignore the underlying HSM architecture and file distribution. The basic strategy of Scatter Gather is **Just Plain Copy**. Variations are to copy the entire system all at once or one chunk at a time, usually by directory. With one chunk at a time, planning must be done to avoid frequent mount table changes. Both of these variations require the system to be healthy and take a very long time, because they churn the system at all of the different HSM levels.

The definition of Systematic for this paper is to *not* cross media boundaries by using a knowledge of the underlying HSM architecture. This implies a layered approach scoped within migration levels. Systematic file selection tends to be much faster because it deliberately controls media mounts. It can also clear filesystem space permitting the stage-in of files at deeper migration levels without triggering new migrations to make space available.

Approaches are tailored to the level being evacuated. Any approach at one migration level may appear Scatter Gather at other levels. The main strength of a Systematic approach is that the HSM system is *not* required to be healthy. Non-healthy systems can have the healthy parts evacuated first and this can improve the health of the system. The Systematic approach can skip over the non-healthy parts of an HSM system, permitting creative [NON-Front-Door] approaches to be used for this "inaccessible data."

User Data Availability

Taking the system **Off-Line** is often the first strategy considered. On some non-HSM systems, the outage window required to relocate all of the data is small enough to be acceptable. This is the standard dump-restore paradigm not covered by this paper. Very large, very critical, or HSM systems require too much time to copy to be able to relocate the data in an outage window short enough for their uses or users.

Putting the system in **Read-Only** mode is a **Semi Off-Line** strategy. This prevents new data from being added to the system. However, it does not end migration churning on HSM systems, because users continue to access their own files as a part of their regular usage. It also means a significant change in work process for some sites which use their filesystems as a primary working area.

Leaving both the old and new storage **Read-Write** is a **Full On-Line** strategy and is the primary focus of this paper. New data can be created during the relocation process, and the work process is minimally impacted. In particular, this strategy was used while developers were actively running make in their filesystems. An example of moving a critical filesystem would be relocating /usr [describing the freeing of blocks for running programs is left as an exercise for the reader]. This technique can be used to eliminate downtime, or to turn a long outage window into a quick reboot. On the other end of the spectrum, some

HSM systems can take months to relocate their data which is why a single outage window is unacceptable.

Algorithm Descriptions

Just Plain Copy

The basic mechanism is to use a standard copying tool like cpio, tar or dump, and replace entire directories or sections with symbolic links as each directory or section is completed. This is similar enough to common Systems Administration practice that no pseudo code is presented.

Pros

- "Really Easy"

Cons

- *Very* slow (jukebox trashing, filesystem churning).
- Requires **Read-Only** or **Off-Line** mode (*downtime*).
- Since process is slow:
  - Requires new mount points or remounting of new storage on old mount point.
  - Requires user retraining for changed mount points.
  - Requires user education regarding access to old files.

Forward Relocation

The basic mechanism is to replace files on the old storage with symbolic links pointing to the new storage after each file has been copied. The authors have used this option on all of the non-HSM filesystems in the case studies.

Pros

- **No downtime**
- New storage doesn't require pre-population
- Don't have to reboot any clients before starting

Cons

- Directory collapses can lead to stale NFS handles
- New files in non-collapsed directories written to old storage
- Hard Linked files left for last
- Special handling required for emacs and mh

Forward Relocation Algorithm Pseudo Code

- *Force Flag*
- *Unconditional Flag*
- emacs *problem*
- mh *problem*

```
find source objects on old storage
    # this may mean a list of files
    # just staged-in on HSM Systems
while ( source )
does source not exist?
    ignore it and get next object
        # pointless to relocate nothing
is source a relocation link?
    check to see if it has been renamed
    (basename of link text != basename)
        # might be emacs, mh or mv...
        if renaming wouldn't overwrite
        existing file on dest,
            rename destination
if source is a directory
    use cpio to duplicate it
    if duplication successful,
        set ownership and permissions
    loop until no further changes:
        renames_needed = 0
        compare all relocation links
        (link text !~ basename)
            renames_needed++
            if wouldn't overwrite
                rename dest.
    end loop
    if renames_needed != 0
        report error:
            link renaming problem
        get next object
    is directory empty
    or only leave behind links?
    yes: if collapse flag is set
        collapse it with rm -rf
        and replace with
        leave behind link,
    no: ignore it
        and get next object
general check for all remaining types
    if target exists and not force flag
        ignore it and get next object
        # prevent overwrites
        # helps with "emacs"
        # and "mh" problem
    if source older than target
    and not unconditional flag
        ignore it and get next object
        # prevent double relocation
if source is a symbolic link
    is it actually safe to relocate it?
        # two pages of discussion
        # and comments in the code
    yes:    relocate it with cpio
        and replace with link
    no:     ignore it
        and get next object
if source is a file
    if ignore migration flag
        copy and replace
    else, is file migrated?
        ignore it
        and get next object
```

```
    if link_count > 1
        if name not in inode table
            record inode num and
            name in inode table
        count instances in inode
        table for inode number
        ( in_table >= link_count )
        yes:    names listed valid?
            copy and replace all
            remove from inode tbl
        no:     get next object
any other file type
    use cpio to duplicate it
    and replace with leave behind link
end while
```

Reverse Relocation

The basic mechanism is to pre-populate the new storage with the tree of *directories* without copying any files and make symbolic links pointing to the *files* on the old storage. While it is not required, transparent access to the new storage can be provided by remounting the old storage on a different mount point and mounting the new storage on the original mount point. Finally, the links are replaced on the new storage with files from the old storage. Because there is a potentially large outage window during the pre-population, the authors have only utilized this method on systems that can be placed in an **Off-Line** or **Read-Only** state. However, following pre-population and client reboots as necessary to remount the new storage, the systems can be returned to the **Full On-line** state.

Pros

- New files written to new storage.
- Access to new storage does *not* go through old storage.
- Handles emacs and mh problem better.

Cons

- Requires pre-population of entire filesystem with directories and symbolic links (*really hard with 500,000+ files to do in one outage window.*)
- bu Requires outage window or read-only during pre-population with symbolic links.
- Requires Client reboot if new storage is mounted transparently on old mount points.
- Requires User training if new storage is on a new mount point.
- Not transparent to user (extra outage window or read-only).
- Hard Linked files left for last.
- Open files may produce stale NFS handles.

Reverse Relocation Algorithm Pseudo Code

- Assume primary user reference point is through new storage
- Then new storage has true names

```
duplicate directories only on new storage
make symbolic links in new storage
    pointing to files on old storage
find reverse links on new storage
    # this may mean a list of files
    # just staged-in on HSM Systems
while ( rev_link )
if rev_link points to a symbolic link
    is link a relocation leave behind?
    no: if safe,
        pull it over
        using rev_link's name
        and delete it
    yes:    delete it
if rev_link points to a file
    is file migrated?
        if not force copy,
            get next rev_link
    if link_count > 1
        if src_name not in inode table
            record src_inode num,
            src_name, and rev_link
            in inode table
        count instances in inode
        table for src_inode number
        ( in_table >= link_count )
        yes: src_names listed valid?
            rev_links listed valid?
            pull over all
            using rev_link names
            and delete
            remove from inode tbl
        no:     get next rev_link
        # this might leave
        # a few behind, but
        # the inode table
        # shows which ones...
    pull it over
    using rev_link's name
    and delete it
if rev_link points to a directory
    # we really should not see any
    # directories passed to us
    # but we can handle them...
    if error_on_directory
        report an error
        get next rev_link
    is src directory is empty
    or only leave behind links?
    yes:    collapse it with rm -rf
    no:     get next rev_link
any other file type
    pull it over
```

```
    using rev_link's name
    and delete it
end while
```

Hybrid Relocation

The basic mechanism is to use **Forward Relocation** for all rapid access media and switch to **Reverse Relocation** for slower media. The Pros and Cons and pseudo code are as described in the two cases above. This is useful for very large HSM systems, or ones that have more than one migration level.

Another **Hybrid** technique uses a modified **Reverse Relocation** which acts more like a **Forward Relocation** in that it leaves behind forward relocation links on the old storage when it replaces the reverse links on the new storage. This technique can be used to permit valid access through both the old and new storage and allows a **Reverse Relocation** to be used without remounting the new storage on the old mount point. However, having both paths available to users can be somewhat problematic and this technique eliminates the inherent emacs and mh compatibility of **Reverse Relocation**. [*The code starts to look very much like* **Forward Relocation**.]

Pitfalls

Double Relocation Problem

When doing multiple passes over the source filesystems, careful checking must be done to avoid relocating relocation links destroying valid data on the destination storage by creating self referential relocation links. This is especially the case when directories are being deleted and folded into single symbolic links, where it is easy to cross into the new storage without noticing. The actual scripts or programs used must check for this at several points.

Pathological Filename Problem

With files created by PCs, Macs and GUI applications, filenames that contain shell special characters have become common. (In this case, white space is considered a shell special character since it is a field separator.) These can be handled in a number of ways. If there is a small number of such files, find them first and correct their names in place. If there are no "quote" characters, try to protect them from being interpreted by the shell. A more general approach is to use STDIO instead of command line parameters to pass all filenames. This last suggestion works for everything except filenames that have embedded carriage returns, newlines or nulls.

emacs and mh

These applications present special challenges because they rename files rather than reusing the inode on the other end of a symbolic link. At some sites, front end interfaces to mh have an additional behavior that makes use of **Forward Relocation** difficult: these front end programs fork and change their working directories to the mh directories. When these directories are collapsed they become symbolic links and the front end programs exit. Thus, in some cases, it is best to do directory collapsing when users are logged off.

If mh only deleted files by renaming them, that would not be too bad, but it then reuses the filenames it has cleared up. The effect is that mh renames its files for most operations and will desynchronize the two filesystems. To keep the filesystems synchronized, an additional step must be taken, and the relocation worker process must not relocate newer versions of the files with the same names.

cpio under SunOS

On SunOS, cpio always returns a zero exit status, so its exit status cannot be used from within scripts or programs. The System V version of cpio does not have this problem.

Post Copy Checksumming

Post copy checksumming using md5 would be a good feature to implement on unreliable networks and for the justifiably paranoid.

Inode Creation Optimizations

For better performance, each symbolic link can be created prior to directory collapse or file copy and mv used to shift it into place following removal. (mv is slightly faster than creating an inode with ln -s.)

End Game

Forward Relocation

When relocations have completed, the source filesystem will have been collapsed down to a single or at most a few symbolic links for top level directories. The final goal is to have the new filesystem mount from the same point as the old filesystem. This is where even **Forward Relocation** may require some client outage.

In an automounted environment where the filesystem is occasionally quiescent on the client (not held open by some process on the client) the automounter can be made to perform the remounting of the new storage on the old mount point transparently. Changing the automount tables (and signaling the automounters on the clients) to have the new storage mounted both on the new or temporary mount point and the original mount point of the old storage will cause the client machines to use the new storage exclusively the next time they access and remount the original reference mount point.

A day or a week later, most of the clients will have ceased using the old storage. Those clients that have not remounted the new storage on the old mount point can be determined by inspection of the old server's showmount output. Direct intervention can sometimes be done on the client: killing the process that has the old storage open and waiting for the automounter to unmount it before restarting the process avoids a client reboot. If this fails, those few clients

that remain can be rebooted, but *no server outage is required.* Once the old storage has been unmounted from all clients, it can be taken off-line.

In statically mounted environments, remounting can also be done one client at a time. With a bit of skill and luck, only a few clients will require reboots.

Reverse Relocation

At completion, the new storage will have no reverse links left in it. Any files left on the old storage are probably abandoned (deleted on new storage) or replaced by newer versions on the new storage.

If the new storage was mounted on the old mount point before relocation was started, then the desired appearance has been attained. If a temporary mount point was used for the new storage then some remounting may be necessary. However, if the new mount point can become a permanent reference, then only an unmounting of the old storage is required. If remounts are required, a client outage may need to be scheduled, and user retraining undone (since the new storage used a *temporary* mount point).

In an automounted environment, the automount tables can be changed and client automounters signaled. With static mounts, clients will have to be individually unmounted from the old storage. As with **Forward Relocation**, the old server's showmount can be used as a starting point for finding clients who need to be unmounted.

Case Studies

Case 1: Northrop-Grumman Corporation

Seminal Case: The Genesis of On-line Relocation

The Northrop-Grumman case was the genesis of **Full On-line Relocation** techniques. On the system there, the multi tiered HSM software had deadlocked at the first HSM level. This level filled to capacity and was unable to migrate files to deeper levels and, lacking space, was unable to retrieve files from the next deeper tier. With the backing store locked up, continuing file creation on the primary media caused them to fill to capacity and refuse to take new data. With the primary media full and the first tier storage deadlocked, no retrievals of staged-out files could be completed. Any client systems that referenced staged-out files or tried to create new files would suffer permanent NFS timeouts (they used hard mounts for robustness) and would eventually hang.

A replacement fileserver large enough to accommodate all of the data was already in place, but there seemed to be no way to relocate the data to it. At that time, over a hundred thousand of the half million files managed by HSM were in this state. On a multi-vendor UNIX LAN of around 200 regular users, at least twenty percent (20%) of the workstations had to be restarted each day to circumvent HSM NFS hangs. Something had to yield.

After a month of trying to repair the system, Freyburger and Cordrey were seeking a way to avoid abandoning all of the inaccessible data. *That* was when the innovation to relocate the files on-line by replacing them with symbolic links was made. At the time, all issues of losing small amounts of data because of race conditions became secondary to recovering as much inaccessible data as possible and resolving client system hangs. All resident files were relocated one at a time to the new server and replaced with symbolic links in the hopes that some of the inaccessible data could be retrieved once space became available on the old storage.

In the first phase, find and the vendor supplied version of ls were used to identify resident files, with cpio being used to relocate them. As it turned out, freeing space on the primary media was enough to relieve the pressure on the HSM system. In the process, more and more files that had previously been inaccessible became available again. It was also necessary to manually recreate the HSM databases several times as the filesystems were evacuated, but that was a well documented process, already in the manuals supplied by the HSM vendor. This phase alone was sufficient to completely evacuate one of the filesystems.

In the second phase, a script was written to iterate through the database for each filesystem and force relocate those staged-out files that were local to the first tier HSM storage. Since only a few dozen files remained when this phase was completed, "fingerprinting" techniques were used to locate, for recovery by hand, those last few files.

No third phase was needed to recover files from the second tier HSM storage because the evacuation was complete. This was despite the fact that the robot was 80% full – all of the data in the robot was stale, representing deleted and prior versions of current files, because garbage collection had never been done.

Race Conditions and Problems

No data was lost to race conditions! Some users even ran make and similar programs in their directories while those directories were being swept clear of files. Since the relocation involved about a half million files in active use by two hundred developers, this came as a pleasant surprise to Freyburger and Cordrey.

One unsurprising anomaly was encountered: some executables (web server daemons in this case) exited when their binaries were relocated.

During the development of the software to do **Full On-line Forward Relocation**, two main problems were encountered: **Double Relocation** and **Pathological Filenames**, both of which are discussed in the previous section.

Non-HSM Filesystem Relocation

There were two filesystems on the old servers that were not HSM managed or had never had files

staged-out. Since the servers were slated for decommissioning, that data had to be relocated as well. One of those two contained several web sites which supported the entire corporation, so it had to be available at all times with no outage.

Having used **Full On-line Forward Relocation** on filesystems under HSM management, the authors applied their software to those normal and critical filesystems. The relocations completed in a single pass, during the production day. Further, because the binaries for the http daemons were stored local to the front end web server which served its content from the NFS mounted volume, the daemons continued serving with no interruptions due to having their binaries moved out from under them.

Case 2: Hughes Space and Communications

As part of a standard technology refresh program, the obsolete, non Y2K compliant Convex 3240 was replaced and the filesystems on it were rehosted to a new file server. Most of the data was destined for a read-only section of the new storage, while home directories were placed in a filesystem with strictly enforced quotas.

Non HSM File Systems

One filesystem had never staged-out files to HSM tapes. This filesystem was evacuated to the read-only "legacy" disk space in one pass using **Forward Relocation** during the production day.

Home directories were handled differently. Since large numbers of project files had been stored under home directories, most of those files were destined for read-only storage. Only dot files and dot directories were relocated to the new, read-write, home directory storage. Similar to a **Reverse Relocation**, directories and files appearing in the top level of each user's home directory were pre-populated with reverse links pointing to the old storage. The old storage was set to read-only mode by management request. After this, the users were free to replace those links with actual files; the reverse links had been created to reduce the impact of home directory relocation.

HSM File Systems

The system was still working, but the media and tape drives were aging and failing. It was also too slow for users to relocate their own data. Due to the extremely large number of files (approximately 997,000) pre-populating the new storage with that number of symbolic links was not practical. Therefore, to minimize user impact, relocation of resident files was started using **Forward Relocation**.

Shortly after the data movement began, management requested that the old system be placed into a read-only state. This changed the availability state to **Semi On-line**, and data relocation continued using the **Forward Relocation** algorithm.

Relocation was paused after the resident files were completed. The Convex was taken **Off-line** and the primary reference point for the user community became the new "legacy" mount point. Shortly thereafter, the HSM system was rehosted on a physically smaller system.

Following rehosting, the new *old* server was placed **On-line** in **Read-Only** mode and **Reverse Relocation** links were populated into the "legacy" storage. This permitted users to read copies of their files even before they were relocated, reducing the burden of by hand recovery and relocation.

Data movement resumed using **Reverse Relocation** by retrieving all files on a particular medium and feeding their names to the relocation worker program. Some repair of damaged tapes was done and files from the repaired tapes were retrieved. Backup tapes were also used to retrieve files whose HSM tapes had degraded beyond usability.

On the new storage (a read-only legacy filesystem), files older than about one year were archived to DLT tape. Once archived, these files on the new storage were replaced with symbolic links pointing to the nonexistent object, archive, so that users browsing the filesystem would be able to view the names of all files available.

Case 3: RAND

Non HSM Home Directories: Forward Relocation by Request

All UNIX account home directories (over a thousand) resided on non-Y2K compliant servers, which had to be upgraded as a part of a standard technology refresh program. The replacement servers were separate fileservers with large RAID boxes at each campus.

mh is used pervasively at RAND, thus, because the resynchronization of the source and destination file systems was not built into Version 2 of the **Forward Relocation** algorithm, it was not used to move UNIX account home directories to the new servers.

However, users could request an early relocation of their home directories by contacting their help desk. As part of this standard Help Desk procedure, **Full On-line Forward Relocation** *is* used to move their home directory to one of the new servers.

*HSM systems: **Enhanced** Just Plain Copy*

This HSM exit was accomplished by Weissler. It is included for completeness to demonstrate that highly successful **File System Relocations** do not require **Forward** or **Reverse Relocation**.

RAND acquired two Epoch optical hierarchical storage management systems in 1989-1990. The initial systems were Sun 4/75 workstations with a proprietary Epoch operating system based upon SunOS 4.0.3. Epoch used Ingres as the supporting relational database with Hewlett-Packard and Hitachi optical Jukeboxes populated with WORM [write once, read many] media. A series of upgrades brought the

systems up to Sun Sparc 20 workstations running SunOS 4.1.3 with erasable optical media.

By 1996, it was clear that the systems would have to be replaced. Backups had become increasingly difficult as the amount of data increased: it was common for a full backup to run several days, rendering it of questionable integrity. Staff turnover left RAND with little expertise in HSM which in turn led to the deterioration of administration. On going garbage collection efforts decreased with the staff turnover, resulting in many of the 1200+ optical media being under 50% utilized. The vendor stopped supporting the non-Y2K compliant hardware which rendered relocation of the data mandatory.

The system was running, healthy, old, and slow. Because it was healthy, a PR campaign was necessary. Some users were convinced to buy their own disks, some wanted the "higher performance" of not having to wait for stage-ins, and others had to be shown the lower overall support cost of newer technology storage systems.

Analysis was conducted to find usage patterns. Three patterns emerged and data that followed these patterns was copied to different target servers. However, because the replacement servers did not arrive on site at the same time, a systematic draining of each optical media was not possible since only a portion of each media could be relocated to a given server. The first server arrived and a portion of the data was moved, and the process was paused. Relocation temporarily resumed after the second server was delivered. When third server arrived, the last of the data was evacuated, some three months after the process began.

The system was placed in a **Read-Only** or **Semi On-line** state during the relocation. The Epoch utility, epls, worked rapidly enough to allow media preparation in the form of pre-load lists per directory. To avoid thrashing, *all files* were staged-out to optical storage leaving the file systems largely empty. Files in the pre-load lists were then staged-in in bulk using epbsi. Data was then manually copied using tar in relatively small chunks. These improvements on the **Just Plain Copy** technique virtually eliminated jukebox thrashing. This makes this example much higher performance than a brute force **Just Plain Copy** approach.

Summary

The **Forward** and **Reverse** algorithms described in this paper offer a different approach to data relocation that does not appear to be in common use. Since they offer options for providing data availability during the relocation process, there are benefits to be reaped by sites choosing to employ these methods. The intent of the authors is to seed these techniques into the thinking and planning of Systems Administrators and Managers.

Performance Comparisons

The performance of an **On-Line** fileserver is *infinitely* higher than the performance of an **Off-Line** fileserver. While the **On-Line Relocation** methods presented here take longer to run on a fileserver when compared with previously available methods, those previously available methods generally require filesystems to be made unavailable to users during the copy. At sites with high uptime requirements, no comparisons of wall-clock times are relevant.

Since **Reverse Relocation** requires **Read-Only** mode or some downtime, it suffers the same problem as previously available methods, and should only be used at sites that can tolerate these changes in data availability (generally lightly used HSM systems or non-healthy servers).

Non-HSM Uses

Server replacements can be done during the production day using a **Full On-Line** technique. As the availability of large capacity disk systems brings them into wide deployment, the time required to relocate the files from one server to another is becoming longer. With these long duplication times, and availability demands, **Full On-Line Relocation** algorithms can be used to reduce the impact of such transitions by minimizing outage windows and allowing data to be relocated during the production day. These techniques can even be used to relocate files within the same server as would be required to move from a traditional filesystem to a journaled filesystem or to relocate large directory trees. Given the choice of large outage windows or near 100% availability, some administrators can reduce the impact of their relocation projects with techniques similar to these **Forward** and **Reverse** algorithms.

Limitations

These techniques are not appropriate for all filesystem relocations. They fail to prevent stale NFS handle errors for environments where the directories are held open by a process being cd'ed there for long periods of time. Files used by programs that keep them open for a long time and change them regularly, like databases, are likewise inappropriate.

The construction of the programs required to perform these tasks is within the capabilities of Sage Senior (Level 4) administrators. However, caution and forethought must be applied to the construction of the code to avoid the pitfalls of double relocation and pathological filenames.

Ongoing Work

Work is under way to enhance the algorithms by recoding them in C++ with advisory file locking and post copy check summing. This work will also be published with significantly expanded pseudo code and will include a full discussion of when a symbolic link may be safely relocated.

Author Information

Vincent Cordrey <cordrey@acm.org> first experienced UNIX in 1981 on a PDP 11/45 running Version 7. He did Systems Administration and wrote custom business software from 1984 through 1987, porting the solution to UNIX in 1988. That was when his work became exclusively UNIX Systems Administration.

Doug Freyburger <freyburger@ieee.org> started in the computer industry in 1978, working on projects from custom VLSI design for spacecraft at the Jet Propulsion Laboratory to stereoscopic video games for a start-up. In 1986, after doing Systems Administration as a sideline for five years, he switched to doing it full time, and has been at it ever since.

Jordan Schwartz <jordan@colltech.com> started his career in data processing as a third shift computer operator at RAND in 1989, and was promoted to the Systems Administration group in 1993. He has been consultant with Collective Technologies since 1998.

Liza Weissler <liza@colltech.com> worked as a technical writer at Systems Development Corporation and RAND, but moved to Systems Administration at RAND in 1987 when she decided it was more interesting to do things rather than write about them. She joined Collective Technologies in 1999.

ServiceTrak Meets NLOG/NMAP

Jon Finke – Rensselaer Polytechnic Institute

ABSTRACT

Network port scanning tools can be helpful in mapping services and exposures, but in large environments you often get more information than you can handle. This paper describes a project to take the output from NMAP/NLOG and merge it with the existing enterprise host management system. This makes it simple for service or platform specific administrators to study the machines in their purview.[1]

Introduction

Due to some recent security problems at our site where a number of machines were cracked due to exposures in in.statd, we decided to step up our project of removing or disabling unneeded services on the workstations and servers maintained by our department. Unfortunately, the press of other projects prevented our system administrators responsible for the various operating system (AIX, Solaris, and IRIX) from working on the investigation. So rather than try to look at these systems from the inside, we decided to try looking at them from the "outside".

NLOG and NMAP

At about the time that this project started, our networking department had installed NLOG [5] and NMAP [4] and had started scanning all of our address space. This meant that we had all of the raw data we needed on ports (services), and we just needed to break it down and display it so that we could understand it effectively. This method has the added advantage of being able to scan machines that might not be directly supported by the staff in our department.

To quote from the NMAP web page, "nmap is a utility for port scanning large networks." It is actually a collection of modules that can scan TCP, UDP and ICMP services, and has facilities for bypassing firewalls and packet filters, as well as techniques to reduce the chance of scans being detected. While the stealth features are not of interest to us, the general port scanning facility is quite useful. Another interesting feature of NMAP is identifying the operating system via "TCP/IP fingerprinting". By sending a number of special packets with assorted options set (and other techniques), NMAP is often able to identify what operating system, and sometimes even the version of the operating system that is running on the target computer based on the responses to the test packets.

There is a related package called NLOG that works with NMAP. The NLOG home page states "Nlog is a set of perl scripts that help you manage your NMAP log files. Included are conversion scripts, a CGI Interface, and the documentation to build your own analysis applications." Although the NLOG package offered some nice query options, it is somewhat limited by not being able to break down the information into the format that we needed. My group supports a number of public access Unix workstations, as well as some, but not all, of the computers owned by institute faculty and researchers, spread over a number of different buildings, subnets and domains. Thus a simple sort by IP address or domain is not enough for our needs. It is also nice to know who owns and administers each computer, and NLOG doesn't have that information readily available.

Service Trak

In 1997, we started a project called *ServiceTrak* [3] to assist in documenting our meta-system configuration. ServiceTrak generates a number of web pages of system configuration and server information. This seemed to be a natural place to hang the port scan information. We added the services (ports) discovered by NMAP to each of system (computer) pages, and added a new tree of discovered services, with all of the systems (hostnames) providing that service. One of the neat features in NMAP is to identify the operating system being run by the target system. It was interesting to see how this related to what we thought was there.

Since the actual port scans were being done by another department, I was only concerned with the database portion of the project. This broke down into four parts; loading the raw NLOG data into Oracle, comparing the raw data to the previous scans, defining some controls and "safety levels" on services and finally generating some web pages.

Converting and Comparing

The process of comparing new port scan data to our existing port scan information is closely coupled to the Oracle load process. One of the things we want to maintain is some historical record of what ports were open on a host, even if they are no longer open. This can help to track what problems and unexpected services that have been cleaned up. To this end, we want to keep track of when we last checked for a host or port, as well as when we last found something.

[1] Normally, I would use the term "domain", but in many cases, the machines in question are NOT grouped by internet domain.

Raw Data Load

The NLOG database format is intended to be processed by PERL scripts, so that it lends itself to easy reformatting. One of the utilities in the NLOG package converts NMAP scan files into NLOG ".db" files. For each host, it lists the IP Address, the number of ports found, the actual port list and NMAPs best guess at the OS of the target host. The port list includes each of the port numbers and the associated protocol (TCP/UDP) that it found for that host.

Our first step is to get the raw data from NLOG loaded into Oracle. This will allow the rest of the processing to take place inside the Oracle database server.

These files are processed by a shell script, which figures out the scan date for each file from the file system (ls -lt), pushes the contents of the .db files through awk to generate one .sql[2] file for each .db file. The script also generates a master .sql file that invokes each of the generated .sql files and then calls the rest of the processing. All together, this collection of scripts and generated scripts manages to load all of the raw data into the database and handles the housekeeping details. The generated data into the Raw Host Data table (see Table 1) and the port information into the Raw Port Data table. After the raw data is loaded, the Domain_Id field is filled in using the Oct_1-Oct_4 values in the Hostmaster [1] IP Address table[3]. The Domain_Id plays an important part in comparing the raw data with the historic data.

[2]The Oracle command line interface, SQL*PLUS will read .sql files for oracle commands and expressions.

[3]All of our Hostmaster (DNS) data are maintained and stored in our Oracle database. The Domain_Id is an internal, unique identifier for each host at our site.

Comparing Raw Host Data To The Historical Data

Now that we have all of the raw host and port data split up and loaded into the database, we need to compare it with our historical data. Rather than pulling the data back out of the database to compare it, we wrote a set of PL/SQL[4] procedures. The original scans are done as a range of one or more subnets, so a given set of data files to be processed will have some, but not all of the hosts on our network.

Preparing and Retrieving the Host Data

Extracting the raw host data is easy. Since we are interested in processing all of the raw data we have, a simple database select is possible. In the processing code (see appendix 1), we define the following cursor[5]:

```
Cursor Raw_Host_Scan is
Select IP_Add, OS_Type,
       Load_date, Domain_Id,
       Rowid
  from Raw_Host_Data
 order by domain_id;
```

Although the historic data (stored in the NMAP_System_List, see Table 2) is derived from raw host data, we don't need to include all of the raw data. Since we can obtain the IP address from Domain_Id, we don't need to include it here. We have added some additional date fields to keep track of past scans.

[4]PL/SQL is Oracle's procedural extension to the SQL Standard. It allows for programs to be executed on the database server which can yield performance gains, as well as storing the code in the database itself.

[5]A cursor allows you to define a query in PL/SQL

Name	Type	Size	Description
IP_Add	Varchar2	32	IP Address of host. This is the primary key.
Oct_1	num	3	First Octet of the IP address. Breaking this out simplifies some database operations.
Oct_2	num	3	Second Octet of the IP address.
Oct_3	num	3	Third Octet of the IP address.
Oct_4	num	3	Forth Octet of the IP address.
OS_Type	Varchar2	255	The type of operating system as determined/guessed by NMAP
Load_Date	Date		Date we loaded this record.
Domain_Id	num	9	Internal identifier to match with Hostmaster tables.

Table 1: Raw_Host_Data Oracle table definition.

Name	Type	Size	Description
Domain_Id	num	9	Internal identifier to match with Hostmaster tables. This can be used to get the hostname and the IP address.
OS_Type	Varchar2	255	The type of operating system as determined/guessed by NMAP
First_Detected	Date		When we first detected this IP address.
Last_Updated	Date		When we most recently found something at this address.
Last_Checked	Date		When we last looked on this subnet.

Table 2: NMAP_System_List Oracle dable definition.

However, since we may not be processing all subnets[6], we just want to work on the historical records for the subnets found in the current set of raw records. To do this, we define the Historic_Host_Scan cursor:

```
Cursor Historic_Host_Scan is
Select nsl.Domain_Id, nsl.Rowid
  from Nmap_System_List nsl,
       dns_ip_address dia
 where nsl.domain_id =
       dia.domain_id
   and dia.ip_octet_3 in
        (select distinct oct_3
           from Raw_Host_Data)
 order by nsl.domain_id;
```

This query will get the all of the previous NMAP scan entries we have processed on this set of subnets[7]. This basically has four steps. The first step is the sub query Select distinct oct_3 from Raw_Host_Data, which will create a list of all the different subnets found in the raw data. The next step is to identify all registered IP addresses in the DNS_Ip_Address table and return those Domain_Id s. This list of Domain_Id s is then matched against our historic host scan returning the Domain_Id and Rowid of the historic record. Finally, this list is sorted by the Domain_Id . The Rowid is an internal Oracle identifier for that specific entry in the database. It can be used for rapid (efficient) access to that row.

Processing the Host Data

We now have two lists, each sorted by the Domain_Id : the "raw" data from the most recent scan, and related historic data. By comparing the Domain_Id s while stepping through both lists together, we can detect newly found (scanned) hosts, historic hosts that were not in the latest scan, and hosts that are in both lists. (The PL/SQL source code is available in Appendix 1.)

If it "raw" Domain_Id is lower, that means we found a new system that was not previously in the historic data. We insert this record into the database and get the next entry in the "raw" list. (If there are no other changes, the lists should be "in synch" again.) If the two values are the same, the records are for the same host and we need to compare the port values (see the next section).

If the "raw" Domain_Id is higher than the Domain_Id from the historic record, then the record in the historic list represents a previously known host that was not detected in the most recent scan. There are several possible reasons for this: the host may have been retired, there could have been a network problem, or the host may be down for some reason. We want to maintain the historic data for this host, including the fact that it was not found. We do this by updating the Last_Checked field in the historic data. (We don't do anything with the ports, since we don't have any data.) This condition will be identified by the reporting tools. The historic list is then advanced to the next record, and the process repeats until the lists are in synch again.

Consider the example data in Table 3. There are both raw and historic data for sam.rpi.edu (Domain_Id 7), so we update the port information and then get the next raw and next historic Domain_Id s. Now we find that the raw id (12 - george.rpi.edu) is less than the historic id (15 - fred.rpi.edu), so we know that george.rpi.edu is a new record, and we insert it. We then select the next raw entry where we get fred.rpi.edu (15) which matches the existing historic entry. These two match, so we update the ports and get the next two entries in the list. This time, the historic entry dave.rpi.edu is less than the raw entry, sharon.rpi.edu. Since we didn't detect dave.rpi.edu in the most recent NMAP scan, so we need to update the Last_Checked_Field it and then get the next historic entry, sharon.rpi.edu, which matches the raw entry and we do the port compare and we are done.

Raw Id	Historic Id	Name
7	7	sam.rpi.edu
12		george.rpi.edu
15	15	fred.rpi.edu
	22	dave.rpi.edu
25	25	sharon.rpi.edu

Table 3: Example data.

Handling Port Data

We also need to keep track of what ports (services) were encountered for each host. To this end, we have a raw port table (Table 4) and a historic port table (Table 5). As with the historic host information, we

[6]The NMAP scans are done on a "Class C" subnet basis, even if target host subnet is different. This makes it easier for the people running NMAP, and makes the processing easier.

[7]All of the IP address information happens to be stored in this same database, making this selection on IP address very easy. Alternately, the subnet information could also be stored in the NMAP_System_List table.

Name	Type	Size	Description
IP_Add	Varchar2	32	IP Address of host. This is the primary key.
Port	Num		Port number.
Family	Varchar2	12	Port type such as "TCP" or "UDP".
Load_Date	Date		Date we loaded this record.

Table 4: NMAP_Raw_Ports Oracle table definition.

don't need to keep track of the IP Address in the port table, instead we can use the Domain_Id to associate the ports with the host.

For each host we are updating or inserting, we select the ports to go along with the raw and historic records. For the raw records, we use the IP address as the key, and for the historic records, we use the domain_id as the key. We order the records from both sets by port and family, and again loop through, comparing ports and families, adding, deleting and updating records as needed. To get the raw port data, we use the following cursor, using the address value to limit the ports to just the ones for the current host (address) in question:

```
Cursor Raw_Port_Scan
      (address varchar2) is
Select IP_Add, Port, Family,
     Domain_Id, load_date,rowid
  from Nmap_Raw_Ports
 where IP_Add = address
 order by Port, Family;
```

In a similar way, we get the historic port information for this particular host, using the Domain_Id as the key:

```
Cursor Cooked_Port_Scan
       (Did number) is
Select Port, Family, rowid
  from Nmap_Port_List
 where domain_id = Did
 order by Port, Family;
```

Both of these selections are sorted by the Port, and then by the address family. These two lists are then compared, stepping through them, much like we did with the host comparison.

Controls

When we display ports (services), it would be nice to include a bit more information about the port than simply the service name. As a starting point, we loaded */etc/services* into a table. While this gives us a service name and sometimes a brief description of the service, we added another column called "safety". Since the primary objective of this project was to improve security, we rated each service as to how "safe" it is to have running. In some cases, while the named service might be considered safe, common security exploits may be using that port as a back door into a system. For these cases, seeing that this port active might be a good indicator that a computer had been hacked. In other cases, some versions of that service may have known holes, so additional checking may be required to determine if the computer in

Name	Type	Size	Description
Domain_Id	num	9	Internal identifier to match with Hostmaster tables. This can be used to get the hostname and the IP address.
Port	Num		Port number.
Family	Varchar2	12	Port type such as "TCP" or "UDP".
First_Detected	Date		When we first detected this IP address.
Last_Updated	Date		When we most recently found something at this address.
Last_Checked	Date		When we last looked on this subnet.

Table 5: NMAP_Historic_Ports Oracle table definition.

Type	Description
Safe	Service is safe to have running
Server	Ok for server class machines, questionable for desktops
NoCrypt	This service exposes sensitive information to network sniffing
Suspect	Often indicates a hacked machine
Unknown	We haven't decided yet.

Table 6: Safety Validation Table NMAP_Service_Danger_Types.

Port	Fam	Name	Safety	Description
9	tcp	discard	safe	Discard packets
21	tcp	ftp	server	
22	tcp	ssh	safe	Secure shell
23	tcp	telnet	NoCrypt	Remote access
31	tcp	msg-auth	Unknown	MSG Authentication
1524	tcp	ingreslock	Suspect	ingres

Table 7: Services NMAP_Service_List.

question is running a safe version of the service. Other protocols are inherently insecure and we may want to discourage their use. We defined the safety values[8] listed in Table 6.

The more challenging job of course, was to rate the services as to their safety level, and expand the descriptions where needed. Naturally, our safety ratings reflect our department's own needs and policies. A partial list is shown in Table 7.

One of our security objectives is to eliminate clear text passwords on the network. Thus, we flag "telnet" as a somewhat dangerous service. We also want folks to use central FTP servers, so that we mark FTP as "server". While we don't have anything against Ingres, we don't run it on our machines, and the ingres lock port was recently used in a particular exploit script. Machines with the ingres lock port open tend be hacked, rather than offering Ingres. Therefor we mark ingress lock as "suspect". We have not classified all services (our service file listed 924), rather we are concentrating on classifying the ones that are showing up on machines we are interested in.

[8]One nice thing about Oracle is the ability to define validation tables that define what values a column can have.

The last control element we need, is some way to define how we want to group hosts in the output pages. We already have a number of host groups set up in our database, these are used for access control (like generating /etc/hosts.lpd), usage statistics [2] and so on. Thus, we had a start of the host groupings that we could use. This resulted in another table, Service_Track_NMAP_List.

This table is used in the generation of the top level of the NMAP web tree. At present, it allows breakdowns by host group and a dump of all hosts with detected services.

Outputs

The top of the NMAP web tree looks something like Figure 1[9]. There is a standard heading with logos and other boilerplate that we omit from this paper. Each of name entries is a link to a tree of ports for that group. A typical host group web page looks something like what is shown in Figure 2. Again, the standard heading is omitted.

[9]In general, the generated web pages are simply HTML tables. Rather than taking screen shots from a web browser, we have reproduced the tables here. HyperLinks are indicated by italic.

Name	Type	Size	Description
Name	Varchar2	32	The name used for grouping.
Tag	Varchar2	16	Used in filenames in the web tree.
Group_Id	Number	12	Points to the top level of the host group "branch".
Description	Varchar2	255	A short description to be included in the web pages.

Table 8: Service_Track_NMAP_List Oracle table definition..

Name	Hosts	Ports	Description
All	3222	365	All scanned hosts
CIS Staff Workstations	60	35	computer center staff Unix workstations
Remote Access	4	19	Public Access Timeshare machines
CUSSP	217	53	Faculty Unix Support Program machines
RCS Workstations	284	45	Public Unix Workstations
NIC Cluster	62	101	High Performance Computing Cluster

Figure 1: NMAP Services.

Service Sample Page
ssh (22/tcp)
Remote Access NMAP Ports
• Secure Shell • Safety Level: Safe • Tab Separated Values Page

Hostname	First Detected	Last Detected	Last Checked	OS
cortez.sss.rpi.edu	12-May-99	12-May-99	12-May-99	AIX 4.2
rcs-sun1.rpi.edu	12-May-99	12-May-99	12-May-99	Solaris 2.6-2.7
vcmr-12.rcs..rpi.edu	12-May-99	12-May-99	12-May-99	Solaris 2.6-2.7
vcmr-19.rcs..rpi.edu	12-May-99	12-May-99	12-May-99	

Figure 1a: Service Sample Page

From the host group page, we can continue to drill down the tree, selecting one of the services of interest. As seen in Figure 3, we add a little more to the basic table, along with "danger level" and description, we include a link to a Tab Separated Values Page. The TSV file has an entry for each of the systems listed, including useful information such as owner, sys admin, location, and a lot of other information that is already in the Service Trak database. This has proven very useful, since it can easily be loaded into a spreadsheet. Service Trak generates a TSV file for just about any page the has a list of computers.

We can continue to drill deeper. Each hostname listed is a link to that host's entry in the Service Trak tree. We have expanded that entry to include what ports were detected on that host by NMAP. Of course, you can click on those service names and end back up near the top of the NMAP tree.

Conclusions and Surprises

The first reaction of most of the staff to see the pages was "cool", followed almost immediately by "Hmm, I didn't know we were running that there, I will have to look in to that." In that, the primary objective of the project was achieved. It also pointed out a number of anomalies in our configurations. For instance, our remote access machines are all supposed to provide the same services. However, it was quickly noticed that the "host counts" for services did not match.

The last column in the list of hosts for a particular service, is NMAPs guess at what the operating system being used on that machine. We were somewhat disturbed to find some of our "trusted host" Unix machines (trusted for LPR) were in fact running Windows NT[10].

We also added the NMAP "guess" of the operating system to our hinfo program. The hinfo program takes a host name or IP address, and returns owner, admin and change history and anything else we have in the database for a machine on our network. This program is frequently used by our NOC staff when investigating oddities.

For security and privacy reasons, we do not allow public access to the Service Trak web pages. Although much of the information is available to the public, and NMAP is easy to install, we don't want to make things too easy for people who want to attack our site. The web pages are on a protected web server, limiting access by either IP address (the staff subnet) or via ID and Password authentication.

At present, we only output information via the web. However, since many of the Service Trak pages have a TSV option, staff interested in other formats can quickly download the page to a spreadsheet and produce reports in other formats.

Service Trak does have a list of "sanity checks" it runs on a daily basis, and we could add checks to look for changes or "unsafe" NMAP entries and generate an alert. However, we are not making regular NMAP scans. Each scan takes a day or so (this is

[10]The disturbing part was that we still trusted them, not that they were running NT. At present, our NT machines do not use our Unix name space, and "administrator" is NOT a valid Unix user!

Name	Port	Family	Danger	Count	Description
echo	7	tcp	safe	4	
discard	9	tcp	safe	4	
daytime	13	tcp	safe	4	
chargen	19	tcp	Unknown	4	
ftp	21	tcp	server	2	
ssh	22	tcp	safe	4	Secure shell
telnet	23	tcp	NoCrypt	4	
time	37	tcp	safe	4	
sunrpc	111	tcp	Unknown	4	
ident	113	tcp	safe	2	
loc-srv	135	tcp	Unknown	2	Location Service
smux	199	tcp	Unknown	2	snmpd smux port
exec	512	tcp	Unknown	4	
login	513	tcp	Unknown	4	
shell	514	tcp	Unknown	4	no passwords used
printer	515	tcp	Unknown	4	line printer spooler
uucp	540	tcp	Unknown	1	uucp daemon
ta-rauth	601	tcp	Unknown	2	For AFS kerberized services
writesrv	2401	tcp	Unknown	2	Temporary Port Number

Figure 2: RCS Remote Access Services.

mostly how we are doing it, my understanding is that NMAP can run a lot faster) and we have not resolved all of the policy issues with scanning the network. Since there are tools other than Service Trak may be more appropriate for detecting problems in real time, we have not pursued this avenue.

NMAP Quirks

We noticed that the NMAP scan appears to kill inetd on some platforms. This was later confirmed in some bug reports. These reports also mentioned some other systems that might react poorly to getting scanned. We certainly noticed some other system administrators who reacted poorly to being scanned. We are now exploring some of the policy implications of port scanning (by our own staff) as well as ways of running NMAP in less "destructive" modes. We have also removed some of our major network gateways from the scans (at the request of our network operations staff), as the scans generated a number of SNMP traps that generally indicate hostile activity.

Alternatives

Clearly, most places do not run ServiceTrak, and many do not even use a relational database to manage their system configuration. However, the thing that gave us the biggest boost, was the ability to look at systems based on our host groupings. The NLOG package may allow for simple integration of your own site configuration information. This would be a good avenue for exploration. Given that we already had ServiceTrak in operation, we did not explore expanding and enhancing NLOG.

At present, the Service Trak "output" is a huge tree of web pages that are regenerated on a daily or weekly basis. This regeneration takes a few hours to run, generates 34,000 files, and uses 120Mb of disk space. While disk space is still pretty cheap, I expect that some parts of Service Trak will by moving to dynamic web pages, generated on demand. Still, a lot less things have to work in order to make files available versus dynamic web pages, so I expect that at least part of Service Trak will remain in files, regenerated daily.

References and Availability

All source code for the Simon system is available on the web (or via AFS). See http://www.rpi.edu/campus/rpi/simon/README.simon for details. In addition, all of the Oracle table definitions as well as PL/SQL package source are available at http://www.rpi.edu/campus/rpi/simon/misc/Tables/simon.Index.htm.

Acknowledgments

I would like to thank David Hudson of Network Support Services, Rensselaer Polytechnic Institute for his work with installing and running NMAP and NLOG. I also wish to thank David Parter of the Computer Science department of the University of Wisconsin for his comments and suggestions on all the drafts of this paper.

Author Information

Jon Finke graduated from Rensselaer in 1983, where he had provided microcomputer support and communications programming, with a BS-ECSE. He continued as a full time staff member in the computer center. From PC communications, he moved into mainframe communications and networking, and then on to Unix support, including a stint in the Nysernet Network Information Center. A charter member of the Workstation Support Group he took over printing development and support and later inherited the Simon project, which has been his primary focus for the past eight years. He is currently a Senior Systems Programmer in the Server Support Services department at Rensselaer, where he continues integrating Simon with the rest of the Institute information systems, and also deals with information security concerns.

Reach him via USMail at RPI; VCC 319; 110 8th St; Troy, NY 12180-3590. Reach him electronically at <finkej@rpi>.edu. Find out more via http://www.rpi.edu/~finkej.

References

[1] Jon Finke. Simon system management: Hostmaster and beyond. In *Proceedings of Community Workshop '92*, Troy, NY, June 1992. Rensselear Polytechnic Institute. Paper 3-7.

[2] Jon Finke. Monitoring usage of workstations with a relational database. In *USENIX Systems Administration (LISA VIII) Conference Proceedings*, pages 149-158. Rensselear Polytechnic Institute, USENIX, September 1994. San Diego, CA.

[3] Jon Finke. Automation of site configuration management. In *The Eleventh Systems Administration Conference (LISA 97) Proceedings*, page Unknown. Rensselaer Polytechnic Institute, USENIX, October 1997. San Diego, CA.

[4] Fyodor. Nmap free security scanner. Web Site, 1999. http://www.insecure.org/nmap.

[5] HD Moore. Nlog: Nmap log management tools. Web Site, 1999. http://nlog.ings.com.

Appendix 1 – PL/SQL Process host data

```
procedure Add_Host ( Raw in Raw_Host_Scan%Rowtype ) is
begin
        Insert into Nmap_System_List
               (System_Id, Domain_Id, Os,
                 First_Detected, Last_Updated, Last_Checked)
        values (Raw.System_Id, Raw.Domain_Id, Raw.Os_Type,
                Raw.Load_date, Raw.Load_date, Raw.Load_date);
        Update_Port_List(Raw);
        Delete from Nmap_Raw_Host_List
         where rowid=raw.rowid;
end Add_Host;
procedure Delete_Host ( Historic in Historic_Host_Scan%Rowtype ) is
begin
        Update Nmap_System_List
           set last_checked = Global_Check_Date
         where rowid = historic.rowid;
end Delete_Host;
procedure Update_Host( Raw in Raw_Host_Scan%Rowtype,
                       Historic in Historic_Host_Scan%Rowtype) is
begin
        Update Nmap_System_List
           set System_Id = Raw.System_Id,
               last_checked = raw.load_date,
               last_updated = raw.load_date
         where rowid = historic.rowid;
        Update_Port_List(Raw);
        Delete from Nmap_Raw_Host_List
         where rowid=raw.rowid;
end Update_Host;
procedure Compare_Host_Lists
is
        Raw     Raw_Host_Scan%Rowtype;
        historic  Historic_Host_Scan%RowType;
begin
Open Raw_Host_Scan;
Open Historic_Host_Scan;
Fetch Raw_Host_Scan into Raw;   -- Prime the pump
Fetch Historic_Host_Scan into Historic;     -- and this one
loop
   exit when Raw_Host_Scan%NotFound and Historic_Host_Scan%NotFound;
   <<InnerLoop>> loop
   if Raw_Host_Scan%NotFound then
      Delete_Host(Historic);
      Fetch Historic_Host_Scan into Historic;     -- and this one
        exit InnerLoop;          -- continue
   end if;
   if Historic_Host_Scan%NotFound then
   then
         Add_Host(Raw);
         Fetch Raw_Host_Scan into Raw;
         exit InnerLoop;          -- continue
   end if;
   -- Ok, we know they are BOTH here.... Lets compare
   if Raw.Domain_Id > Historic.Domain_Id then     -- Raw list MISSED a historic,
         Delete_Host(Historic);
         Fetch Historic_Host_Scan into Historic;     -- and this one
```

```
        exit InnerLoop;          -- continue
    end if;
    -- Lets try the other way
    if Raw.Domain_Id < Historic.Domain_Id then     -- New Raw record
        Add_Host(Raw);
        Fetch Raw_Host_Scan into Raw;
        exit InnerLoop;          -- continue
    end if;
    -- Not greater, not less, must be the same
    Update_Host(Raw, Historic);  -- Do the ports
    fetch Raw_Host_Scan into raw;
    fetch Historic_Host_Scan into historic;
    exit InnerLoop;          -- InnerLoop is just for a continue function
  end loop InnerLoop;
end loop;
```

Burt: The Backup and Recovery Tool

Eric Melski – Scriptics Corporation

ABSTRACT

Burt is a freely distributed parallel network backup system written at the University of Wisconsin, Madison. It is designed to backup large heterogeneous networks. It uses the Tcl scripting language and standard backup programs like *dump*(1) and GNUTar to enable backups of a wide variety of data sources, including UNIX and Windows NT workstations, AFS based storage, and others. It also uses Tcl for the creation of the user interface, giving the system administrator great flexibility in customizing the system. Burt supports parallel backups to ensure high backup speeds, and checksums to ensure data integrity. The principal contribution of Burt is that it provides a powerful I/O engine within the context of a flexible scripting language; this combination enables graceful solutions to many problems associated with backups of large installations. At our site, we use Burt to backup data from 350 workstations and from our AFS servers, a total of approximately 900 GB every two weeks.

Introduction

In the past several years, network installations have become increasingly large and complex. These networks consist of more workstations, and those workstations, particularly in research environments, run a wider variety of operating systems. In addition, the recent dramatic decrease in the price of hard disk drives has made it feasible for users to store large amounts of data on their workstations. Many users also want the best of both the *datafull* (all data stored on local disks) and the *dataless* (all data stored on file-servers) workstation models. Finally, system administrators are under constant pressure to decrease the amount of time during which backups run. Together, these points make backups more difficult than ever.

As an example, the University of Wisconsin, Madison, Computer Sciences Department has over 350 workstations, each of which runs one of seven different flavors of UNIX. Some of these workstations have eight gigabytes of data or more stored locally. In addition, we have over 500 gigabytes of data stored on our AFS [4] fileservers. We are given only six hours each night during which to perform backups.

Burt, the "Backup and Recovery Tool," was developed to address these issues. We had several specific goals in mind when we developed Burt, some of which are common to many backup systems:

- Store data to long-term media
- Provide a means by which to track stored data
- Retrieve stored data
- Ensure the integrity and reliability of stored data, to the degree allowed by the storage technology
- Ensure the speedy backup and recovery of data, to the degree allowed by the storage technology
- Provide a mechanism to automate the backup of data
- Secure backups, to a reasonable degree, from likely attacks
- Support a variety of storage technologies
- Support network backups
- Provide an easy-to-use interface

Other goals were more specific to our needs. These related to the quantity and distribution of data at our site, and the heterogeneity of our installation:

- Support the backup of large aggregate amounts of data, on the order of hundreds of gigabytes, and gracefully scale to accommodate growth.
- Support the backup of large amounts of data from a single source; in particular, allow for the backup of atomic sources that are larger than the capacity of a single tape or other storage element.
- Support the backup of data from a large number of sources, on the order of tens of thousands of sources, and gracefully scale to accommodate growth.
- Support backups of data from many different kinds of sources; in particular, support backups of multiple flavors of UNIX workstations and of AFS data.
- Allow easy integration of future kinds of data sources; in particular, support backups of all future operating systems and architectures with minimal changes to the backup system.

The backup system we had been using, DK [12], did not meet these needs adequately. We determined that the best way to address all of these needs was to construct a new backup system. We built that system in two layers, one compiled and one scripted, using C for the compiled language and Tcl [10] for the scripted language. This model has been used in many modern applications, such as word processors and other office software. We found that it could be applied to backup software as well, where it gave us both performance and flexibility. In particular, the Burt I/O engine and supporting scripts provide:

- Support for the backup of a wide variety of data sources, and easy integration of new types of sources, by using standard backup programs like *dump*(1) and others

- Fast backups, often at or near the maximum speed of the storage device
- Support for large atomic sources and large aggregate amounts of data
- Support for user-interfaces tailored to a particular site
- Easily extended functionality, through the Tcl/Tk core and extensions

This paper discusses the design of Burt and its features as seen by a system administrator. We will begin with a brief overview of what a backup system is. The next two sections describe the Burt Architecture and user interface. The next section contain comparisions to other backup systems. The penultimate section describes our experiences with Burt at the University of Wisconsin, Madison, over the past two years. Finally, the last section discusses our future plans.

Overview

A *backup system* consists of many subsystems. At the coarsest level, those subsystems may be grouped into a hardware component and a software component. We will largely ignore the hardware component for the remainder of this paper.

The software component may be further divided into an agent responsible for moving data to and from the hardware, a mechanism for tracking what data is on each backup media volume, and a user interface. By itself, the Burt engine is only the first of these: a data moving agent. The other pieces are Tcl scripts that a system administrator creates to suit the needs of a particular site. These scripts handle all the other software portions of a backup system, including tracking data and providing a user interface.

With this in mind, Burt is designed to enable a system administrator to backup a large collection of data, from a large number of dissimilar data sources to a single output destination. In the context of this paper, *data source* includes disk partitions on workstations, AFS volumes, individual files, databases, and any other unit of data for which a data stream can be created. The output can be written to any location, including files, sockets, and tape drives.

The Burt Architecture

Burt consists of two components: a compiled component, and Tcl script based *backup types*, which are bound to the compiled component at run time. We chose Tcl as the scripting language for Burt because it is easy to extend, and we felt that it was easy to learn. In addition, the availability of the Tk graphical toolkit extension assured us that we would be able to easily make graphical user interfaces for our system.

The Engine

The compiled component, which we will refer to as the engine, extends the scripting language Tcl [10] and consists of roughly 7,500 lines of C. A special effort was made to adhere to POSIX.1 standards [6] to ensure that the engine could be easily ported to many UNIX systems; we have successfully ported Burt to Solaris, Linux, and SunOS. When loaded into a Tcl interpreter, the engine adds a few commands that serve as an API for controlling the engine:

- backup, for initiating backups
- recover, for initiating recoveries (restores)
- schedule, for scheduling data sources for backup or
- readtape, for verifying checksums on tape and building a list of data sources on tape
- status, for obtaining runtime status information

The engine has a number of significant features, but two are particularly important. First is has multiplexing capabilities, and second, it is "dumb."

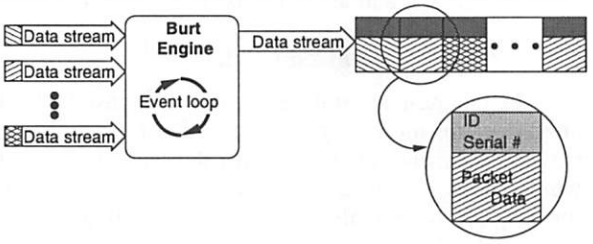


Figure 1: Data flow in Burt during backup, as seen by the engine.

The Multiplexing Engine

A key feature of the Burt engine is its multiplexing capability: it receives as input some number of data streams, packetizes and checksums the data, and outputs a single stream containing those packets (Figure 1). Each packet contains a header that indicates the origin of the data, the sequence number of the packet, and other information; and a block of data from an input stream.

The ability to multiplex is important because it enables the system to achieve much higher overall performance than is generally possible when using a single input stream. Other authors have quantified the performance benefit [2, 11]; one paper showed a better-than-linear speed increase as the number of input data streams increased [2]. Essentially, the speed increase comes from the system's ability to compensate for slow input streams by reading data from other input streams – the more input streams in use, the more likely it is that any one of them will have data waiting to be read at a given instant. Multiplexing does add complexity to the system, but we feel that the performance benefits far outweigh the cost of that complexity.

Of course, the engine has demultiplexing capabilities as well. When reading a tape written by Burt, the engine receives a data stream composed of interleaved data from many data sources. The engine only extracts data for those data sources that an operator is interested in recovering from tape (Figure 2).

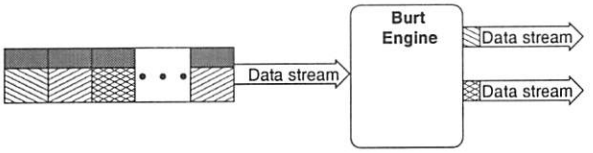


Figure 2: Data flow in Burt during recovery, as seen by the engine.

The "Dumb" Engine

The second critical feature of the engine is that it is "dumb." It does not know where the input data is coming from, nor where it is going to. The knowledge about where the data is coming from is encapsulated in *backup types*, the second component of the Burt engine. The knowledge about where the data is going to is managed by the user interface, and is specified by the operator or administrator. This separation allows Burt to support a wide variety of data sources and storage devices, because the administrator need not change the compiled portion of the system to add support for any type of data source or storage device. The administrator only needs to extend or update the Tcl based backup types or user interface, as described later.

Other Important Features

A number of other engine features are worthy of mention. One such feature is that it writes directly to the tape, instead of writing to an intermediary storage location first, as some systems do [2]. This allows Burt to reduce significantly the total time required to perform a backup.

The engine computes a checksum for every packet written to tape, and writes the checksum out with the packet. The checksum enables Burt to verify that the data read from the tape is identical to the data written to the tape. This provides protection from media corruption. If the engine finds an incorrect checksum when reading the tape, it notifies the operator. The engine uses a 32-bit extension of the Fletcher checksum [3], which we chose because the algorithm is fast, highly reliable, and easy to implement.

The engine allows backups to span tapes. This is particularly important for data sources that are larger than the capacity of a single backup tape. It also allows the operator to append data to a tape previously written by Burt. This can be useful at sites that have aggregate data sizes smaller than the capacity of a single backup tape. Those sites can put backups from several successive days on a single tape, rather than wasting the additional space.

Finally, the engine uses filemarks on tape to signify the start of data from each data source. These filemarks can be used later to fast-forward the tape directly to the point at which the requested data is located. This can greatly improve the time required to recover a particular data source from tape, because the engine can skip all of the data preceeding the relevant data rather than reading it. This is merely an optimization; without the filemark information, the engine can still recover all the data from the tape, but it will take more time to do so.

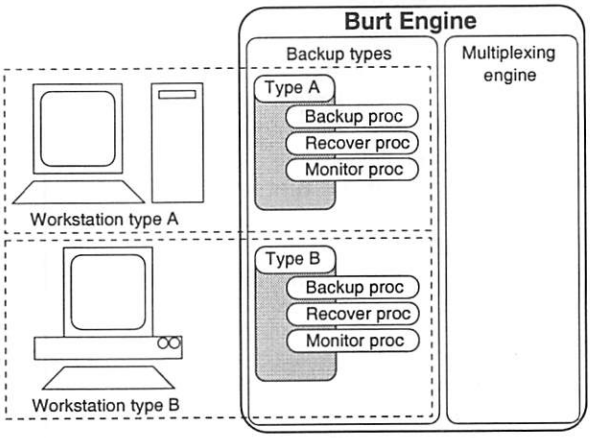


Figure 2: Each type of system has a unique backup type implementation, but every backup type presents the same interface to the engine.

Backup Types

The second component of the Burt engine is the backup types. These are Tcl functions that serve as an interface between the engine and a particular type of data source. A site will have one backup type for each type of data source. For example, we use eight backup types: one for each flavor of UNIX used at our site, and one for AFS. In the terminology of object oriented programming, Burt has a single backup type interface, and each type of data source has a unique implementation of that interface. Figure 2 illustrates the concept.

A backup type consists of several Tcl functions that encapsulate the knowledge about how to backup a type of data source. Burt requires that a backup type implement three functions: backup, monitor, and recover. Each function is *registered* with the engine, meaning that it is associated with a particular type of data source, and is then called by the engine as needed at runtime. For reference, the *solaris* backup type, which we use to backup the majority of our workstations, is included in Appendix 1.

The Backup Function

The backup function is responsible for connecting to and initiating backups of a particular data source. When the engine is requested to backup a data source, it invokes the backup function that is registered for the type of that data source. The engine passes parameters to the backup function that indicate the particular data source to be backed up. The function initiates a backup, typically by exploiting the native backup program of the data source. For example, the backup of a UNIX workstation could use *rsh*(1) to connect to the workstation, and *dump*(1) to perform the backup of a disk partition. The backup function also prepares the standard error output from the native backup program for processing by the

engine. Finally, it returns two data streams to the engine, one along which the backup output from the native backup program is transmitted, and one along which the standard error output is transmitted.

The Monitor Function

The monitor function is responsible for examining the standard error output. As the backup of the data source proceeds, the engine feeds all standard error output received into the monitor function registered for the type of the data source. If the monitor function indicates that an error has occurred, the engine terminates the backup of the data source in question and logs that action along with the error message.

The Recovery Function

The recovery function is responsible for creating a data stream to which recovered data will be written. When the engine is requested to perform a recovery of previously backed up data from a particular data source, it calls the recover function registered for the type of that data source. The engine passes the recovery function information indicating the original origin of the data being recovered, and the recovery function creates a data stream to which the engine can write recovered data. As the engine reads data from the backup media, it writes each packet that is from the requested data source to the data stream created by the recovery function.

Typically, the data stream will be a standard UNIX pipe directly into the native recovery program of the data source from which the recovered data was backed up. Alternatively, the data may be written to a file, which is useful when the operator needs to browse the data and does not want to reread the tape.

Flexibility of the Backup Type Architecture

The backup type architecture is one place where the benefits of separating the system into a compiled and a scripted component are clear. Because the backup types are not compiled, they are easy to modify and extend, and can even be altered at runtime. Certain features of Tcl make it very suitable for use in this context; in particular, the language's sophisticated process control capabilities and extensive string and regular expression operations make it easy to construct backup types for a wide variety of data sources.

The philosophy of exploiting the native backup and restore programs of a particular data source is also essential to Burt's ability to support many kinds of systems. This type of setup enables the administrator to leverage existing tools rather than requiring them to create or maintain programs for performing backups. In addition, it reduces the amount of time between the introduction of a new type of data source and the introduction of backup system support for that data source. The administrator does not need to install anything on the client, provided it already has a native mechanism for performing backups.

In practice, it has proven to be easy to add support for new kinds of data sources. When we originally began using Burt in August 1997, we supported only five varieties of UNIX at our site; since that time, we have added support for two other varieties. In both cases, adding support consisted of less than twenty minutes of work creating a new backup type, a few tests to verify the proper function of the new backup type, and then bringing the backup type "online." An administrator who is reasonably comfortable programming in Tcl should have no difficulty creating or modifying backup types.

The User Interface

In a sense, the user interface is what brings it all together: the engine, the backup types, and the operator. The engine adds a few commands to a standard Tcl interpreter; the user interface is responsible for making use of these new commands to make backups and recoveries happen. This is the second area where the separation of the system into a compiled and a scripted componant has proven to be a great benefit. Because the interface is composed of scripts, it can be easily altered. Perhaps even more importantly, it is easy for each administrator to customize the interface to the particular needs of their installation. Tcl, along with the Tk graphical toolkit, is a good choice for this task. It is easy to create sophisticated graphical user interfaces with Tcl/Tk, a fact which is demonstrated by the wide variety of programs that use Tcl/Tk to create their user interface.

As with backup types, an administrator who is reasonably comfortable with Tcl should have no trouble making user interface scripts. At minimum, these scripts should have some means of starting backups of some list of items, some means of starting recoveries of some list of items, and some means of finding the tape on which a particular item is stored. At our site, we have chosen to follow the UNIX model of breaking a system into many small components. Thus we have separate scripts for performing backups, recoveries, and searches. We also have a variety of "utility" scripts, including runtime status displays and easy-to-use schedule list editors. These scripts range from simple batch-oriented programs to fancy graphical, interactive user interfaces. In total, we have written roughly 3,000 lines of Tcl/Tk code.

The Backup Script

A typical backup script begins by determining what data sources are to be backed up to a given tape. At our site, this is accomplished through statically generated *group* files, which are simply lists of data sources. The operator or batch processor specifies which group file should be read. The script then schedules each data source with the engine, via the `schedule` command. Next, it must register the backup type functions for each type that has been scheduled. Then, the script must open an output stream, typically

a tape drive; again, at our site, the operator or batch processor specifies which tape drive to use. Finally, the script instructs the engine to begin backups, via the backup command.

Once the backups of all scheduled data sources have completed, the script must make a record of the data sources that have been written to the tape, in order for the administrator to be able to locate data from particular data sources. The normal way of doing this is via the readtape command, which builds a list of all the items on the tape, as well as tests the checksums for every packet on the tape to verify that the data has been correctly stored. The list is then stored in a database for future reference. Finally, the backup script will mail the administrator with information about the run. Pseudo-code for this entire process might look like this:

```
call queue_data_sources()
call register_backup_types()
output = call open_output_stream()
call start_backups(output)
wait while backups are not complete
backup_statistics = call
                    get_backup_stats()
table_of_contents =
                    call verify_tape()
call close_output_stream(output)
call write_to_database
                   (table_of_contents)
call mail_operator(backup_statistics)
```

Our backup script can be found on the World Wide Web at [7].

The Recover Script

A typical recover script, like a backup script, begins by determining what data sources are to be recovered. We use an interactive script that allows the operator to enter data sources to recover. The script then schedules each of these data sources with the engine, via the schedule command. Next it registers the backup type functions, particularly the recovery functions, for each type that has been scheduled. Then, the script must open an input stream previously written by Burt, typically a tape; the operator specifies which tape drive to use. Finally, the script instructs the engine to begin recoveries, via the recover command.

The engine will first fast-forward past as many filemarks as it is instructed to, and then begin to read each packet on the tape. The use of filemarks is optional in this case; if they are used, they can dramatically reduce the amount of time required to recover data from the tape. If a packet for a scheduled data source is found, the engine writes the data in the packet to the output stream created for that data source by the recovery function. Our recover script can be found on the World Wide Web at [8].

The Search Script

The search script need only provide the administrator with a means for determining what tape contains the data from a particular data source. We store that information in a simple text database; accordingly, our search script is basically a wrapper around *grep*(1).

Our search script provides limited additional functionality, such as allowing the operator to view the log associated with the creation of a particular tape, and determining the physical location of a particular tape. Our search script can be found on the World Wide Web at [9].

Other Scripts

We use a number of other scripts in addition to the backup, recovery, and search scripts. For example, we have a script that reads only the Burt label from a tape and displays it; we have a script that displays the current speed and percentage complete of a running backup; and we have a script that coordinates the start of nightly backups in batch mode. Most of these scripts are between 10 and 300 lines of Tcl/Tk code.

Comparisons To Other Systems

There are several backup systems that aim to fill the same needs as Burt. This raises the question, why did we create something new rather than using an existing solution?

The answer is plain: none of the systems we examined filled all of our needs, or they did not fill our needs as well as we would have liked. The primary contribution of Burt is that it meets the needs established earlier, and we feel it does so better than other systems. With respect to each system, Burt is more flexible, more extensible, more scalable, or a combination of these. It is also less costly than commercial systems, a factor that has a significant impact at some sites.

In this section, we examine other backup systems, highlighting specific similarities and differences. This is not meant to be a comprehensive comparison of backup systems; we have simply chosen a few systems that we believe other administrators will be familiar with. A more complete comparison of backup systems can be found in [1], though that comparison predates Burt. The systems we consider here are

- Amanda, from the University of Maryland [1, 2]
- Legato Networker [5]

Common Features

All of the systems have some features in common. For example, they each support backups of a range of types of data sources, and they each support backups of several data sources in parallel. However, the implementation of these features varies greatly from one system to the next.

Support for Different Types of Data Sources

One feature that is implemented very differently by the systems is their support for different types of data sources. Although they each support backups of a range of types of data sources, that range is significantly different in each product due to the

implementation. Legato Networker uses proprietary backup programs for each type of data source. Consequently, if Legato has not yet created a backup program for a particular type of data source, Networker cannot support it. Presently, Networker's support for various types of data sources is fairly broad, including Windows clients and a variety of flavors of UNIX, but there is no support for AFS backups, which is critical for our site. This implementation can lead to delays between the introduction of a new type of data source and the introduction of support for backups of that type of data source. It should be noted that the use of custom backup programs has at least one important advantage: it allows Networker to create very detailed indices of the contents of each backup tape. These indices can be made at the file level, which makes it easier for administrators to locate a single file on a backup tape.

Amanda uses standard backup programs for each type of data source, including programs like BSD dump and GNUTar. This leaves the system well positioned to support a wide range of systems, provided that they use BSD dump-like or GNUTar-like backup programs. Of course, this includes the majority of systems that an administrator might want to backup, including Windows clients and various UNIX clients. There does not seem to be direct support for AFS backups. We believe that it would be possible to add such support to the system, although it would require editing Amanda's C source code and recompiling.

Burt also uses standard backup programs for each type of data source, but it is not limited to BSD dump-like and GNUTar-like programs. Burt allows the use of any program that writes to standard output as a backup program. Consequently, Burt can backup not only Windows clients and various UNIX clients, but also AFS, and even more exotic types of data sources, such as World Wide Web sites via HTTP sockets. Part of this flexibility comes from the manner in which the backup type layer is implemented. It provides a sufficiently generic interface to the engine that a very wide range of data sources can be used. The other part of the flexibility comes from the use of Tcl as the implementation language for the backup types.

By using standard backup programs, both Amanda and Burt limit themselves, in a sense, to the capabilities of those programs. For example, Networker's proprietary backup programs allow the creation of backup indices that enumerate every file backed up, rather than just listing the partitions or hosts that were backed up. In the case of Amanda and Burt, the creation of such indices is possible only if the underlying backup programs used in the backup system support that sort of index. BSD dump and derivative programs, for example, do not, but GNUTar and others do.

Parallel Backups

The systems also vary in their implementation of parallel backups. Each supports the backup of several systems in parallel, in order to increase backup throughput. Legato Networker and Burt both multiplex backup data directly to the tape drive, as described above. This allows very high backup speeds to be achieved – often at or near the maximum speed of the tape drive – providing that a large enough number of systems are backed up in parallel. However, the tapes created by such a system may be difficult to read without using the backup software to do so.

Amanda, by contrast, uses a "holding disk" to which backups are written in parallel; from that holding disk, the backup data is written serially to tape. This implementation has some advantages. First, it is faster than serially backing up systems directly to tape, as demonstrated in [2]. And it creates a tape that can be read easily with standard UNIX programs like *dd*(1). However, this implementation can be significantly slower than multiplexing directly to tape. This is primarily due to the addition of a "middle-man" to the data path. Each block of backup data received by Amanda must be written to disk, then later read from disk and written to tape. Burt and Networker have only one write, directly to tape, for each block of data received.

Unique Features

Besides their common features, each system has some unique features as well. Amanda's most notable feature is the high degree of automation that it provides to the administrator. It will automatically schedule backups and perform load balancing on backup tapes. It can also protect against accidental tape overwrites, and as noted above, produces tapes that can be read easily using standard UNIX utilities.

Burt's most notable feature is its high degree of flexibility in terms of supported types of data sources and user interface. We have not created a general purpose, graphical user interface for Burt; by design, such an interface is not a part of Burt. We have created a fairly sophisticated administrative interface, but it is highly site-specific, and probably of little use to other sites, except as a demonstrative example. In any case, we feel that the creation of the user interface is best left to the individual administrator, who is best qualified to know what features are needed at their site. Tcl certainly provides the tools to make an impressive user interface. Similarly, we have not created a general purpose backup scheduler, though we have created a scheduler for our AFS backups, and are in the process of developing a scheduler for our workstation backups. In addition to this flexibility, Burt has the ability to distribute backup data over multiple tapes, and it uses checksums to help ensure data integrity.

Networker's most notable feature is its sophisticated user interface. In addition to providing access to the administrative functions of the system, Networker's user interface provides users the ability to request restores from the system. If the backup media is stored in an automatic loader, such as a tape changer

or tape library, Networker can perform user requested recoveries without administrator intervention. Networker also has the ability to distribute backup data over multiple tapes.

Experience and Performance

We began using Burt as our primary backup system in August 1997. After a few initial problems due to bugs in the implementations of our backup types, everything has gone smoothly.

Overview

We backup a total of around 900 gigabytes every two weeks, counting both full backups and incremental backups. We use a collection of thirty 4 mm DDS-2 tape drives for backups, with 90 m DDS tapes, each with an uncompressed capacity of two gigabytes. We have enabled hardware compression on the tape drives to increase the capacity of our tapes. We choose to backup to a large array of relatively low capacity tape drives for three primary reasons. First, it gives us a second layer of parallelism, allowing us to increase our overall throughput. Second, it limits our loss if a single backup tape is damaged – instead of losing 35 gigabytes or more, as we might if we used DLT drives, we can only lose four gigabytes or so. Third, it is inexpensive to replace the drives if they fail.

Obviously we cannot fully backup everything every night. Instead, we use a two week "epoch cycle". Each night, we peform a full (BSD dump level 0) backup of a portion of the data, and incrementally backup the remainder. This totals roughly 60 gigabytes of data each night. This kind of backup policy is sometimes called *compositional* backups, and was easy to implement with Burt and Tcl.

Implementation

Each night a script is run at 1:00 am (via *cron*(1)) to initiate our incremental backups. These occupy about 15 of our 30 tape drives for about 90 minutes. At 3:00 am, the batch processor initiates the epoch backups that have been selected to run that day, on the other 15 tape drives. At 9:00 am, an operator verifies that the nightly backups have completed successfully. If any require an additional tape, the operator loads a new tape in the appropriate tape drive. If any of the nightly backups have failed, which happens occasionally due to media errors, the operator restarts the failed backup. In addition, if any individual data sources have failed to be backed up, due to network errors or problems with the data source, the operator corrects the problem and runs a special "redo" backup for those data sources. Once all of the backups have finished, the operator loads tapes for the next nightly backup run.

Growth

Since Burt's introduction, we have significantly increased the size of our data set, in both total data size and total number of data sources. Originally, we backed up roughly 500 gigabytes every two weeks, from about 9,000 individual data sources, including AFS volumes and workstation disk partitions. Now we backup 900 gigabytes every two weeks, from about 13,000 data sources. We have had no problems scaling the system to accommodate this growth, although we did have to add additional tape drives to handle the increased data size.

Performance

We have had no trouble finishing backups in the six-hour window allotted to us each night. In fact, most of our backups finish well within that window, with the exception of the odd backup that runs over one tape. Backups that run over one tape do not finish until an operator comes in to load a second tape, and lie idle until that time. In general, backup performance with Burt has been quite good, often at or near the maximum possible rate for the drives and media we use. Network bandwidth and the I/O capabilities of the data sources being backed up seem to be the major limiting factors on our backup speed; Burt itself is not the bottleneck.

We do have some performance problems with our AFS backups, but those are due to the manner in which we schedule AFS backups. Our choices were influenced by the desire to be able to provide fast recovery from catastrophic AFS disk failures. One way to reduce the time needed to perform such recoveries is to minimize the number of tapes from which we must recover data. Accordingly, we schedule all the AFS volumes from a single partition on an AFS server to be backed up to a single tape. This minimizes the number of tapes we have to retrieve data from if that partition crashes, but it incurs a penalty on the backup speed. Typically, our AFS backups run at about half the speed of our workstation backups.

Of course, the purpose of doing backups is to be able to recover data. We have no significant difficulties recovering data, and have performed an average of one recovery a day over the past several months. Note that this number includes recoveries performed due to system crashes, system upgrades, user error, and tests to spot check our backups. One incident in particular stands out: in the summer of 1998, we suffered a complete disk crash on one of our AFS servers. We lost an entire disk containing over 300 user home volumes and several gigabytes of data. Thanks to Burt – and the administrator working on the problem – we had the server back up and running with all data restored within 24 hours. The majority of the time was spent waiting for the AFS servers to process the recovered data; reading the data off the tapes only required a couple of hours, and was performed in a single pass.

Overall, Burt has been a great success at our site.

Future Work and Considerations

Even though Burt in its present form has worked well for us, it is clear that there is still room for

improvement. In particular, there are a few features that we would like to add to the engine.

First, we want to add broad support for tape changers. Many sites use tape changers to further automate the backup process. In some cases, Burt can use tape changers, but the support is not explicit, nor does it extend to the wide variety of tape changers available. This task is complicated by the fact that tape changers do not all use the same control interface, and we do not have any tape changers of our own with which to experiment. We recently received a source code contribution to provide tape-changer support for some kinds of tape changers, and will be working to integrate that code.

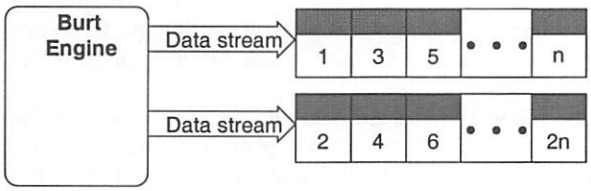


Figure 3 : Sequence

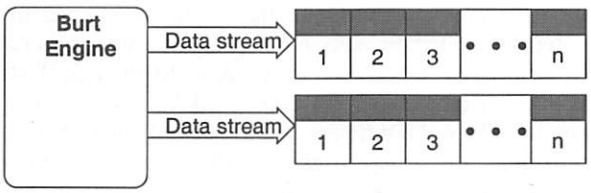


Figure 4: Sequence of writes in mirroring mode.

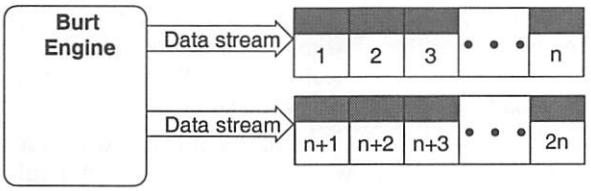


Figure 5: Sequence of writes in rollover mode.

Next, we want to add support for the use of multiple tape drives from a single Burt process. Rather than writing all output to a single tape drive, the engine could write the output to multiple tape drives. Various writing patterns are possible: *striping*, in which writes alternate between tape drives (Figure 3); *mirroring*, in which all data is written to all tape drives to produce multiple copies of the backup tape (Figure 4); and *rollover*, in which the tape drives are used as sort of a "poor man's" tape changer (Figure 5). Each of these has its own benefits and penalties. Striping provides increased throughput, especially for slow tape drives, and increased capacity, but at the cost of an odd tape format. Mirroring provides additional backup reliability, because having two copies of a backup tape guards against media failure, at the cost of some backup throughput. Rollover provides increased capacity, with no significant penalty. Such functionality would be useful in some situations, and would further distinguish Burt from other backup systems.

We would also like to add more explicit support for non tape-like media. In particular, we should better support disk-like media, such as ZIP disks, and various magneto-optical disks. Presently, it is possible to direct Burt's output data stream to a file on disk, but that file is treated like a sequential access construct. It would be best to exploit the features of the disk that make it different from a tape, namely its random-access nature. One simple way to exploit that feature is to replace the use of filemarks to mark the beginning of data from a particular data source with byte offsets to mark those beginnings. We have a number of ideas about how best to implement this and other uses of disk-like media, but have not yet settled on one.

Finally we would like to create an online repository of backup types into which administrators could submit backup types they have created, and download backup types that they may need. We have begun this repository with the backup types that we have created for our site, and look forward to receiving submissions from other administrators.

There is one important consideration that should be mentioned: Windows NT. Many people ask if Burt supports backups on Windows NT. The answer to that question is a qualified yes. Currently, the Burt engine cannot be run on Windows NT. We do not know what is involved with porting the engine to NT, so that capability may be long in coming. However, it is possible to create a backup type to backup data from Windows NT workstations. We know of two methods for doing so. First, the backup type could use Samba [14] to mount the NT volumes on a UNIX system, and then use SMBTAR or GNUTar to backup the data. Second, it could use any of the several Windows NT rsh services available to connect to a remote NT machine, and use GNUTar or *ntdump* (a port of BSD dump to NT that we are developing) to backup the data. We do not use either of these methods presently because neither conforms to our security policies. As soon as we find a way to reconcile that issue, we intend to begin backing up our NT workstations.

Conclusions and Availability

With regard to the goals previously outlined, we feel that we have been quite successful in meeting them. The principal contribution of Burt is that it provides a powerful I/O engine within the context of a flexible scripting language. In particular, Burt provides support for a wide variety of data sources; fast backups; support for data sources larger than the capacity of a single tape; support for large aggregate amounts of data; and extensible interfaces and functionality. Our own experience has shown that Burt provides all of these. We can backup all of the data in

our department, including data sources larger than the capacity of a single tape, and we can do so quickly and reliably. We have been able to easily add support for new types of data sources. We have been able to scale our backup system to accommodate our significant growth in the past two years. We have been able to extend and customize our user interface as need and desired. These are features that all system administrators need from a backup system, and Burt provides them.

In addition, we feel that the model of creating software in two layers, one compiled, for performance reasons, and one scripted, for flexibility and customization purposes, is one that can be applied to a wide variety of problems. As noted, it is a model that has been used in many modern applications, of which office software is a particularly well known example. However, that use has typically been limited to minor customization and automation tasks. We found that a significantly larger portion of the application could be scripted than has been traditionally, and that doing so afforded us an extremely high degree of flexibility. We believe that hybrid applications that feature a substantial scripted component represent the future of software development, and look forward to seeing more software that is made more powerful and customizable by this approach.

Since October 1998, Burt has been available for download from the Burt homepage, http://www.cs.wisc.edu/jmelski/burt. As of mid-April 1999, there have been about 6,000 visitors to the Burt homepage, and about 2,000 downloads of the Burt engine. Currently, we see about 50 downloads of the engine each week. We see about 30 downloads of the documentation each week, which is probably a better indication of the number of people actually interested in using it.

We have begun to receive source code contributions for the Burt engine source code from other Burt users. We believe that this bodes well for the future development of Burt, and look forward to receiving more such contributions in the future.

Author Information

Eric Melski graduated from the University of Wisconsin, Madison in 1999 with a BS in Computer Sciences. While at the university, he worked as a system administrator for four years. Following graduation, he joined Scriptics Corporation in Mountain View, California, where he is a software engineer. Reach him via U.S. Mail at Scriptics Corporation; 2593 Coast Avenue; Mountain View, CA 94043. Reach him electronically at ericm@scriptics.com.

Bibliography

[1] James daSilva and Ólafur Gudmundsson, "The Amanda Network Backup Manager," In *Proceedings of the Seventh Large Installation Systems Administration Conference*. The Usenix Association, November 1993.

[2] James da Silva, Ólafur Gudmundsson, and Daniel Mossé, "Performance of a Parallel Network Backup Manager," In *Proceedings of the Summer 1992 USENIX Technical Conference*, pages 217-225. The Usenix Association, June, 1992.

[3] John Fletcher, "An arithmetic checksum for serial transmissions," *IEEE Transactions on Communications*, 30-1:247-252, January, 1982.

[4] John H. Howard, "An Overview of the Andrews File System," In *Proceedings of the Winter 1988 USENIX Technical Conference*, pages 23-26, The Usenix Association, Winter, 1988.

[5] Legato, Inc., *Legato NetWorker Administrator's Guide, UNIX Version (NetWorker for UNIX 5.5)*, December, 1998.

[6] Donald Lewine, *POSIX Programmer's Guide: Writing Portable UNIX Programs*, O'Reilly &Associates, Inc., March, 1994.

[7] Eric Melski, http://www.cs.wisc.edu/jmelski/burt/lisa99/backup.html .

[8] Eric Melski, http://www.cs.wisc.edu/jmelski/burt/lisa99/recover.html.

[9] Eric Melski, http://www.cs.wisc.edu/jmelski/burt/lisa99/search.html.

[10] John Ousterhout, "Tcl: An embeddable command language," In *Proceedings of the Winter 1990 Usenix Technical Conference*. The Usenix Association, Winter, 1990.

[11] W. Curtis Preston, "Backup Techniques – Dynamic Parallelism," *SysAdmin*, February 1997.

[12] Dan Romike, DK I/O System, March 1987.

[13] J. G. Steiner, B. Clifford Neuman, and J. I. Schiller, "Kerberos: An authentication service for open network systems." In *Proceedings of the Winter 1988 Usenix Conference*, The Usenix Association, February, 1988.

[14] The Samba Team, http://samba.org.

Appendix 1: The Solaris Backup Type

Following is the University of Wisconsin Department of Computer Sciences (UWCS) Solaris backup type definition as used with Burt.

```
##################################################
# Function definitions for solaris dump type
##################################################

##################################################
# solaris_dump
#    The solaris backup type backup proc;
#    initiates a backup of the given atom on the
#    given host at the given level
##################################################
proc solaris_dump {host atom level} {
    global sessionID tmpdir
    set hostlogfile $tmpdir/BURTlog.${sessionID}.$host
    if { [string compare $atom "/"] == 0 } {
        append hostlogfile ".root"
    } else {
        regsub -all {/} $atom "." newatom
        append hostlogfile "$newatom"
    }
    append hostlogfile ".$level.ufs"

    set dumpfd [open "|/s/std/bin/rsh $host -n \
      \"/usr/sbin/ufsdump ${level}uf - $atom\"
       2> $hostlogfile" {RDONLY NONBLOCK}]

    set stderrfd [open $hostlogfile r]
    return [list $dumpfd $stderrfd]
}

##################################################
# solaris_cleanup
#    The solaris backup type cleanup proc; cleans
#    up temporary files created during the backup
#    of the specified host, atom and level
##################################################
proc solaris_cleanup {host atom level} {
    global sessionID tmpdir
    set hostlogfile $tmpdir/BURTlog.${sessionID}.$host
    if { [string compare $atom "/"] == 0 } {
        append hostlogfile ".root"
    } else {
        regsub -all {/} $atom "." newatom
        append hostlogfile "$newatom"
    }
    append hostlogfile ".$level.ufs"

    catch {file delete $hostlogfile}
}

##################################################
# solaris_monitor
#    The solaris backup type monitor proc; checks
#    the given line of dialog for keywords that
#    indicate an error
##################################################
proc solaris_monitor {host atom level line} {
    return [regexp {error|Unknown|EXITED|ATTENTION|abort|Bad} $line]
}
```

```
##################################################
# solaris_recover
#    The solaris backup type recover proc; performs
#    recoveries of the given host, atom and level
##################################################
proc solaris_recover {host atom level} {
    set filename "$host"
    if { [string compare $atom "/"] == 0 } {
        append filename ".root"
    } else {
        regsub -all {/} $atom "." newatom
        append filename "$newatom"
    }
    append filename ".$level.burt"
    return [open "$filename" w]
}
```

This backup type is used to backup disk partitions on Solaris workstations. It includes a backup procedure, a recovery procedure, and monitoring procedure, and a cleanup procedure. Each of these procedures is called by the engine as needed.

The backup procedure is responsible for connecting to the given host and initiating a backup of the given partition at the given level. It uses Kerberos V rsh [13] to make the connection, and uses the Solaris program *ufsdump*(1) to initiate the backup. The standard error output from ufsdump is redirected to a file, from which is it read and passed to the engine. File handles for the backup data and for the standard error data are passed back to the engine.

The cleanup procedure is an optional procedure that is used to perform any cleanup that may be required following the completion of the backup of a particular host and atom. When the engine finds that the backup of something of type *solaris* has finished, it will call the solaris_cleanup procedure with the particular host and atom that has finished. In this case, the cleanup procedure simply removes the temporary file that was used to store the standard error output from the backup of a particular item.

The monitor procedure is responsible for parsing a line of dialog from the standard error output of the backup of a particular item and determining whether or not the dialog indicates that an error has occurred. If the procedure determines that an error has occurred, the engine will abort the backup of the item. In this case, the monitor procedure checks the dialog for certain keywords that indicate an error has occurred.

The recover procedure is responsible for initiating a recovery of a particular item. When the engine finds that it is supposed to recover data from an item of type *solaris*, it will call the solaris_recover procedure with the particular host, atom, and level to be recovered. The procedure returns a writable file handle to the engine, and as the engine reads data from the backup media, if it finds a packet that is from the particular item, it will write the data to the file handle given by the recover procedure. In this case, the recover procedure just writes the data out to a file; our operators must then use the Solaris program ufsrecover to extract the required data from the file.

Design and Implementation of a Failsafe Print System

Giray Pultar – Coubros Consulting LLC

ABSTRACT

This paper describes a printing system designed for an environment with several hundred printers, such that there is no single point of failure, and the print service continues to be available to new print jobs, even if any part of the print system fails.

The paper also describes the following ideas implemented in this system:

- use of a single queue name (myprinter) for almost all print jobs from the client
- use of dynamically created print queues for each user (user-<username>),
- use of dynamically created print queues to manage low cost desktop printers, that are directly attached to a users display device (local-<xterm>),
- integration of the print service with the Zephyr notification system to send print progress messages, error messages, as well as route print jobs based on user location as reported by the notification mechanism.
- ability to route print jobs from legacy systems to all printers in the system, without having to define all the printers on the legacy system. (VM, VMS)

The system was implemented using LPRng [1], taking advantage of its job routing, control file rewriting and dynamic printcap features.

Overview

This paper describes a printing system designed for the research and development division of a pharmaceutical company. Because of its interaction with the FDA (Food and Drug Administration of the United States government), this division produces a large amount of documentation that needs to be printed on short notice. Therefore, printing is one of the top items on the list of "mission critical" systems.

Some of the features of the print system as implemented are:

- Banner pages on colored paper
- Pooling of printers under one queue
- Translation/mapping of usernames from other systems (VM/VMS/NT)
- Graphical user interface for queue management
- Pop-up notices when jobs are queued, being printed, and completed.
- Printing to inkjet printers connected to X terminals.
- printer defaults per user

The Old System

Like most environments, there was an existing print system in place. As much as possible, the team tried to treat the project as a "design from scratch", but in many instances the existing system affected the decisions made in designing the new system.

There were many disadvantages to having an existing system in place: Many users wanted the capabilities of the existing system replicated in the new system, even if these "capabilities" were quirks of the old system. Some users wanted "all" the features of the old system to be available in the new system, even though some of the features were not needed anymore. The technically advanced users had their own ideas about how the system could be fixed; and wanted their (in most cases stop gap) fixes implemented.

On the other hand, there were some advantages of having an existing system: Users were aware of their specific needs that were not being met with the current system; and were able to articulate their requirements for a new system well. Some of the well conceived but poorly implemented features could be re-implemented in the new system so that they would work. Also, the old system could be used as a test lab, or a rapid prototyping lab, to demonstrate some new ideas. It would then be easier to see the users' reactions to these new ideas that would be fully implemented in the new system.

It is therefore difficult to characterize whether the old system was an asset or liability in designing the new system.

Here are some quotes (some made up, others as best as I can recall) from users about the old printing system:

- "I got this printer on my desk 3 weeks ago. When are you going to set it up?"
- "Oh, I sent this job to the printer this morning. I hope I can get it before I go home."
- "I always print twice. One of them never comes out anyway."
- "Why can't I print from application A to printer B?"
- "How was I supposed to know that he was printing 2000 pages. If I had known, I would have used the printer next to it."

- "There are 5 printers in this room; and not a single one is printing my document!"
- "I hate those 'your print job is printed' messages. Can't I turn them off?"
- "The Payroll application can't print on my desktop printer. So every time I print something out, I have to run like mad to the printer room."

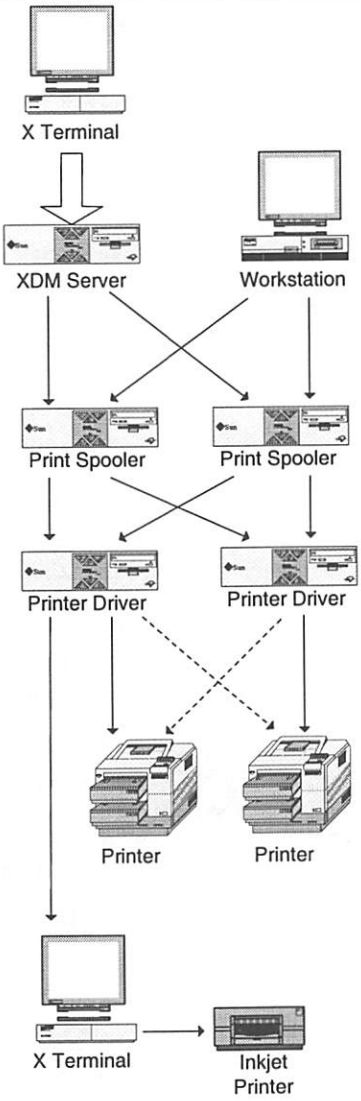


Figure 1: High level Physical design.

The Old Architecture

The old system consisted of two hosts that were dedicated to printing. All servers and workstations were configured with queues that would forward print jobs to server A. Server A would then talk to all the printers and try to send the jobs. Server B was supposed to be a backup for server A. Server B's configuration had not been maintained for a long time: it is unlikely that it could print anything without significant reconfiguration. Moreover, one would have to reconfigure all the workstations and servers to send jobs to server B instead of server A, in case it failed.

Administrators' Problems

From an administrators point, this system was designed to solve the following problems that were being faced:

- How can one manage (create, remove) hundreds of printer queues, if every user wants to have a printer on their desktop?
- How does one set up printer lists in each application, so that users can print to any one of the hundreds of printers available?
- How does one get print jobs from non-Unix systems to print to their printers, with cover pages with Unix login names, and proper accounting, without too much configuration on either the Unix or non-Unix side?
- How does one distribute printer configuration (aka /etc/printcap) information to all their workstations/XDM servers?
- Can one build a print system so that if any of the print servers go down, all printing service can continue?

The Design

The print system consists of two sets of identically configured host(s): The Print Spoolers (PS) and the Printer Drivers (PD).

The purpose of the print spoolers is to receive print jobs from clients, make any user name translations (for non-Unix systems), and/or formatting adjustments (such as landscape, 12 cpi, etc.) and send it to the appropriate PD server.

The purpose of the printer drivers is to communicate with printers and manage the pooling of printers.

A print job from a client (such as a user's workstation) goes through the following steps:

1. client contacts a PS server and sends the job
2. PS server makes any adjustments necessary to the print job
3. PS server sends jobs to PD server.
4. PD server puts job in a pool queue, and waits until a printer in the pool is available.
5. PD server sends jobs to the printer.

This flow is also covered in the section "Life of a Print Job".

Redundancy

As shown in Figure 1, there are several redundant paths a print job can take from a client to the printer. The redundancy of the system is achieved via the mechanisms below.

Client Contacting a PS Server

The clients are configured to send all print jobs to host 'printhost'. This hostname, using DNS, resolves to a round robin list of addresses of all the PS servers.

The client will try all the addresses in turn, until it succeeds in sending the print job.

If a PS server is down, the client will try the next one on the address list. If all the PS servers are down, the lpr client will reject the job at the client rather than queueing the job.

PS Server Contacting PD Server

The queues on PS servers are configured such that each queue is assigned to a deterministic list of PD servers. The PS servers go through the list until they can send the job.

If a PD server is down, the PS server will try the next PD server listed for that queue.

PD Server Choosing a Printer

The printers are arranged in pools. Users will typically submit their jobs to one of these pools. The PD server picks the next available printer in a pool to send the job.

If a printer is down, the PD server (after scheduling one print job that will never complete), will stop sending jobs to that printer as it is not available, and use other printers in the pool.

Failsafe Requirement

The requirement that was stated as: "If a component of the print system fails, the system should continue to function for any NEW jobs submitted to the system. The system will also try to complete as many as the existing print jobs as it can."

The implications of this requirement is discussed in the discussion section.

Detailed Configurations

Client Configuration

The clients that are using this system can be classified in two broad categories:

- hosts with LPRng client software
- hosts with LPD server software

The hosts with LPRng client software are hosts typically under our administrative control. There are two items required on these hosts to get printing working.

1. The lpr binary from the LPRng package
2. The /etc/lpd_client.config file. The main purpose of this file is to indicate where to contact to send print jobs. (e.g., printhost.domain.com). It is unlikely that this file will change. Thus, it is unlikely that the configuration will ever have to be updated on the clients.

Most of the clients not under our administrative control could communicate using the LPD protocol.

```
# $Id: lpd_client.conf,v 1.3 1998/05/05 16:52:19 giray Exp $
default_remote_host=printhost.domain.com.
use_identifier
use_queuename
originate_port 2000 3000
check_for_nonprintable@
```

Listing 1: Sample /etc/lpd_client.conf file.

```
# $Id: printcap.pl,v 1.28 1.3 1998/09/05 13:51:16 giray Exp $
# PS servers
spoolhost:This is a print spooler host:::server:PS
otherspooler:This is another print spooler host:::server:PS
# PD servers
badhost:This is a printer driver server:::server:PD
goodhost:This is another printer driver server:::server:PD
# A pool of printers (aka floor server), floor_0 with the two printers
# rdprint3 and rdprint4, using the PD server badhost, and if badhost
# is unavailable then goodhost
floor_0:Basement Printers:visible:badhost,goodhost:floor:rdprint3,rdprint4
# A HP 4si queue for the printer with hostname rdprint3.domain.com.
# Note that this printer is not directly visible by users, but
# is part of the floor_0 pool.
rdprint3::::hp4si:rdprint3.domain.com
# A remote printer: using LPR/LPD with rp=prllxcsl and
# rm=mvs.cmis.domain.com. Send print jobs to this printer using LPD.
MVS_prllxcsl:Remote prllxcsl on MVS:::remote:prllxcsl@mvs.cmis.domain.com.
```

Listing 2: Sample printer.conf file.

We would typically request the addition of one queue on their hosts pointing to the PS servers.

For example, one would set up a queue on VM that is configured to forward jobs to a queue called "fromvm" on printhost.domain.com. (the print spoolers).

The operation of these queues from legacy systems is covered in the "Legacy Printing" section.

PS and PD Server Configuration

The PS and PD server are set up as typical LPD servers [3], with one major exception: Instead of a static printcap file; the printer configuration information is generated dynamically.

LPRng has the ability to use a script to generate printcap entries instead of using the traditional /etc/printcap file.

For this purpose, we have developed a perl script called /etc/printcap.pl that generates this information using a config file of our creation called printer.conf.

```
floor_0|Basement Printers
:sd=/usr/spool/lpd/floors/floor_0
:as=|/usr/local/lib/print/queued
:bq=floor_0@badhost,goodhost
```

Listing 3: Sample output from /etc/printcap.pl on a PS server.

User Specific Queues

The user queues are queues of the form user-<username>. These queues only exist on PS servers, and are used to route jobs from a user to their selected printers. The main purpose is to make it easy for applications to print: they can print to user-<username>, and the print job will go to the user's selected printer.

This mechanism uses the dynamic printcap feature in LPRng: LPRng will execute the dynamic printcap script /etc/printcap.pl (described earlier) when looking up the printcap entry for user-<username>.

The script, will then look up the user's preferred printer, and return a printcap entry that routes the job to the user's preferred printer. The script, if necessary, create a spooling directory for this user. Therefore, when a new user tries to print for the first time, their queues is automatically set up; and no administrator action is necessary.

If the user has selected "My Desktop Printer" as their preferred printer; then this script will look up the user's location via Zephyr, and route the job to the appropriate local queue (local-<hostname>), described below.

Local Queues

The local queues are queues of the form local-<hostname>. These queues exist both on PS and PD servers, and are used to route jobs from a user to the printer connected to the X terminal that they are logged in on. All the X terminals in this environment are equipped with parallel ports, and several users have connected inkjet printers, mainly for privacy concerns [6, 7].

The main purpose of local queues is to make it easy to route jobs to a terminal that a user is logged in on. Print jobs can be routed to local-<hostname>, and they will go the printer connected to that host.

Similar to user queues, the spooling directories for local queues are created dynamically. As a result, whenever a new X terminal is added, or a printer is attached to an existing X terminal; a queue is automatically created when the first job is queued.

The Most Commonly Used Queue: myprinter

This queue routes its jobs to user-<username> based on the user who submitted the print job. This is accomplished via the routing filter mechanism provided by LPRng.

When a job is sent to "myprinter", LPRng will invoke the routing filter which picks up the username from the control file, and routes the job to user-<username>

DNS Configuration

The system uses multiple address records in DNS to achieve redundancy at the PS level.

An entry for "printhost" is required. This entry will have the addresses of all the PS servers in the system.

X Terminal Configuration

All X Terminals in the environment are configured to accept connections on a specific port, and pass all data received on this port to the parallel port. Using this mechanism, users can connect an inkjet printer to their X terminal.

Support for Other Systems and Legacy Printing

Printing via the LPD protocol:

To allow non-LPRng systems to send jobs to this system, we establish one queue on the legacy system, and one queue on this system for each legacy system that is going to send jobs to this system. This queue is then run through a control file rewrite filter and a routing filter, before being forwarded to the "myprinter" queue.

The control file rewrite filter rewrites the username field in the control file; by doing a lookup of the users username on the foreign system and translating it into their UNIX username. The translation is completely automated, except for the case when the username cannot be found in the mapping table.

There are several reasons for this setup:

- Only one queue on the other system is needed
- The job can be rerouted to any printer
- Using username mapping, proper accounting, and print progress messages are possible.

The major disadvantage, however, is that the username mapping needs to be maintained. It is hoped that in the future, this can be stored in an LDAP style directory.

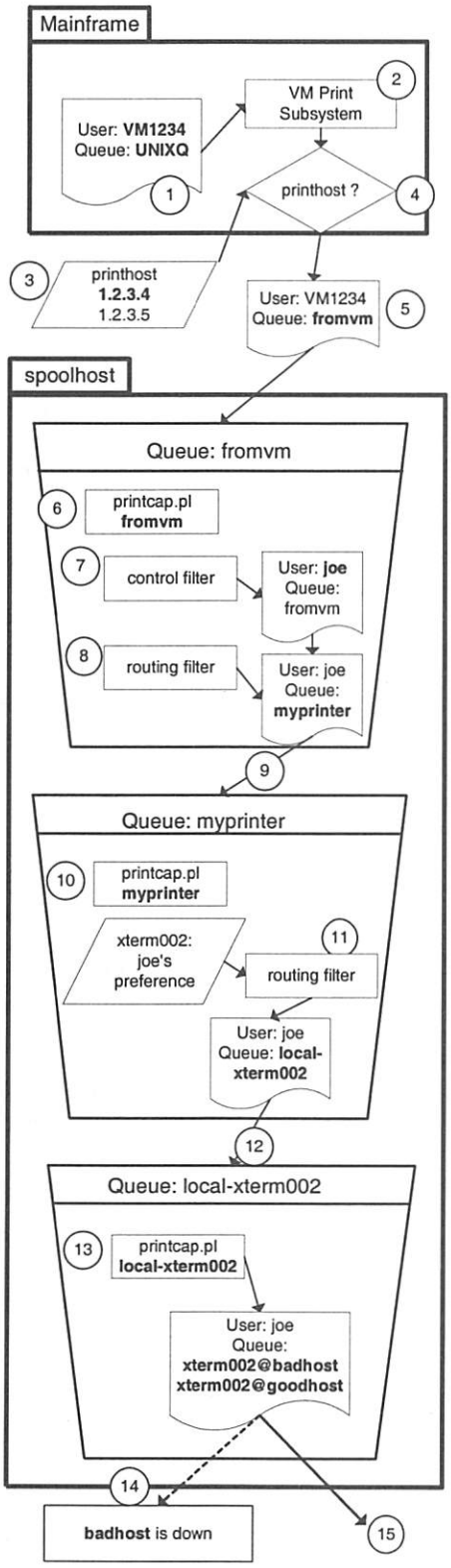


Figure 2a: Logical print job flow.

Printing Via "Port" Functionality in Terminals

Historically, people would use terminals (or dumb terminals) to login to multiuser hosts (e.g., VAX/VMS). Typically, these terminals would also have a printer port that users could attach a printer to.

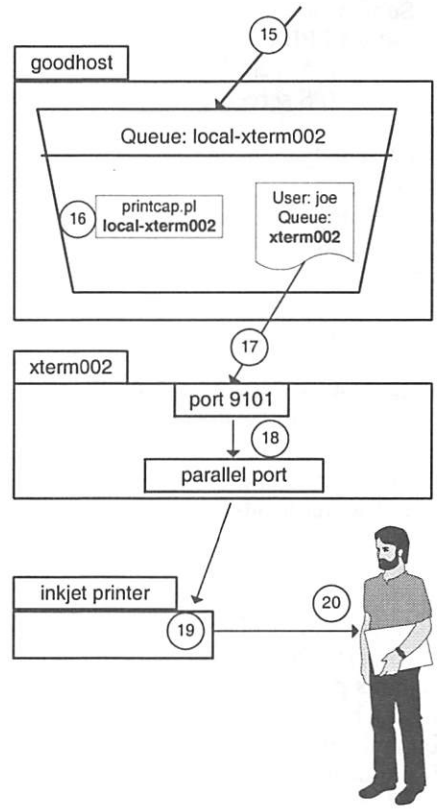


Figure 2b: Logical print job flow.

Instead of having to define all printers connected terminals on a VMS system, they designed a way to print to "any" printer attached to a terminal. The way this works is that the host (the VAX in this case) sends a special sequence of characters that tells the terminal to switch to "printer mode" [4]. Once in "printer mode" the terminal will send all the characters it receives to the printer instead of displaying them on the screen. This continues until the host sends another sequence that tells the terminal to switch back to terminal mode and start displaying the characters.

During a VAX terminal session all characters coming from the VAX are screened using a filter program, that watches for the special sequences that switch into and out of "printer mode".

Once the "printer mode" is switched off, this filter then calls lpr to print all the characters collected.

The Life of a Print Job

This section will describe, in detail, all the processes a sample print job goes through when being printed in order to demonstreate how the different parts of the system work together to get the job printed.

The sample print job is one that originates in the VM system and is destined to a printer connected to an X terminal. In addition, it is assumed that one of the PD servers is down.

- User on VM
- Print Job
- Print Subsystem on VM
- printhost via LPD protocol
- DNS multiple A record
- to spoolhost (PS server) on queue from-VM
- username translation filter
- print job with Unix login
- routing filter
- print job with new destination
- myprinter queue
- routing filter
- print job with user-<username>
- user-<username> queue
- routing filter (look up from settings for desktop printing, look up from Zephyr for location)
- new print job with local-<xterm>
- local-<xterm> queue -
- badhost server local-xterm queue fails
- goodhost server local-xterm queue succeeds
- interface filter
- Xterminal
- parallel port.

1. A user on VM decides to print, invokes the print command and generates a print job.
2. This job is processed via the VM Print subsystem, which will decide to send the job to the "fromvm" queue on "printhost" using the LPD protocol.
3. VM will perform a DNS lookup to resolve the name "printhost". VM will receive a number of addresses associated with this name, and pick one to send the job to.
4. Assume that VM picked the address for spoolhost.
5. VM will then send the job to the lpd daemon running on spoolhost destined for the queue "fromvm".
6. The lpd daemon on spoolhost will execute /etc/printcap.pl to get a printcap entry for "fromvm". This script will generate a printcap entry that lists that both a control file filter and a routing filter will be used. The lpd daemon on spoolhost will then receive the print job and store it in the spooler directory.
7. The lpd daemon on spoolhost will invoke the control file filter associated with this queue. This filter is the username translation filter, which will rewrite the username in the control file, and replace the user's VM username with their Unix login.
8. The lpd daemon on spoolhost will then invoke the routing filter associated with this queue. This filter is the routing filter, which returns "myprinter". The lpd daemon, will redirect this job to the "myprinter" queue.
9. The lpd daemon spoolhost will execute /etc/printcap.pl to get a printcap entry for "myprinter". This script will generate a printcap entry that lists that a routing filter will be used.
10. The lpd daemon will transfer the print job to the "myprinter" queue.
11. The lpd daemon on spoolhost will then invoke the routing filter: This filter will look up the users' preferred printer. Assume that the user has listed "My Desktop Printer" as their default printer. The routing filter will then use "zlocate" to locate the user, and find which X terminal they are logged into. Assume the user is logged in from xterm02. The filter will return local-xterm02.
12. The lpd daemon on spoolhost will execute /etc/printcap.pl to get the printcap entry for local-xterm02. /etc/printcap.pl will create a spooling directory for this queue if necessary. It will return a printcap entry that lists PD servers to try in a particular order: first badhost and then goodhost.
13. The lpd daemon on spoolhost will transfer the print job to "local-xterm02"
14. The lpd daemon on spoolhost will try to connect to badhost. After a while, this connection will time out.
15. The lpd daemon on spoolhost will try the next PD in the list. It will try to connect to goodhost. This time the connection will succeed. The lpd daemon spoolhost will send the job to the "local-xterm02" queue on goodhost.
16. The lpd daemon on goodhost will execute /etc/printcap.pl to get a printcap entry for local-xterm02. printcap.pl will create the spooling directory if necessary. It will also return a printcap entry pointing at port 9101 of host xterm02.
17. The lpd daemon on goodhost will then open a connection to port 9101 of host xterm02 and send the job.
18. The Xterminal xterm02 is configured to send all input from port 9101 to its parallel port. It will send the print job to the parallel port.
19. The printer is connected to the parallel port and and will print the job.
20. The user will pick up their print job, hopefully smiling, and thinking "Hey, this is great. I hit print, and seconds later, here's my print job!" without being aware of all these processes that have gone on in the background.

Features and implementation

This section describes, the features introduced in the overview, and how they are implemented.

Banner Pages On Colored Paper

The banner page program includes the necessary PCL and Postscript code to switch to different paper tray for the cover page and then switch back. The

printers are stocked with colored paper on one tray and white paper on other trays.

Pooling Of Printers

This feature is already implemented in LPRng with subserver queues.

Translation/mapping Of Usernames From Other Systems (VM/VMS/NT)

The translation of usernames from other systems is done using the control file filter feature in LPRng.

Once a job is received, LPRng will execute a filter. This filter programs looks for the username field in the control file, extracts the username, then looks up this username in a simple directory; and finally rewrites the control file with the new username.

Graphical User Interface For Queue Management And Preferences

A simple Tk frontend to LPRng's lpq and lpc was developed. Users can look up the jobs in queue and cancel or move jobs.

This tool also allows the user to change their preferred printer, i.e., the printer that user-<username> will print to.

Pop-up Notices When Jobs Are Queued, Being Printed, And Completed.

This is accomplished via the Zephyr package from MIT's Project Athena. Using Zephyr, it is possible to send pop-up messages to users as their print jobs progress through the printing system. The accounting filters in LPRng are set up to send these Zephyr messages.

Printing To Inkjet Printers Connected To X Terminals.

Printing to inkjet printers connected to X terminals is done via the local-<Xterm> queues mechanism described above.

When the user has selected "My Desktop Printer" as their default destination, their user-<username> queue will route jobs to local-<xterm>, by querying Zephyr to locate the xterminal that the user is logged in on.

For example, for a user "joe" working on xterm02, their queue user-joe would forward jobs to the local queue would be local-xterm02.

The PD server for X terminals will send the job to port 9101 on the host specified: If a user prints a job to local-xterm02, the PD server would contact xterm02 on port 9101 and send the job.

It is possible to print to the workstations using this mechanism as well. All the workstations have a daemon running on port 9101 /usr/bin/localprintd (started through inetd), that will send the jobs to the workstation's parallel port.

Printer Defaults Per User

Another simple Tk application was developed that allows the user to edit their printer preferences. Using this application, they are able to select their preferred printer, orientation and see their username mappings for print jobs received from other systems.

These preferences are stored in a simple flat file, which is used by the PS servers.

Modifications to LPRng

The following modifications, merged into LPRng distribution, were made to LPRng.

Multiple A Records

LPRng only used the first address returned by DNS. The code was modified to try all the addresses returned by the name query. (This is similar behavior to most telnet clients).

With this change, clients using the new LPRng lpr client would try contacting all the Print Spoolers, until it found one that worked.

Multiple "rm" Entries In Printcap

Traditionally, lpd servers could only have one remote machine (rm) entry for each remote queue. This was changed to add the ability to define multiple hosts to be tried in order.

This change was necessary to implement failover between PD servers. Each queue has a list of PD servers that it is hosted on. It was necessary that PS servers to try these hosts in order; for queue listing consistency.

Solutions To Administrators Problems

The section describes how this system solves each of the following administrators' problems.

How does one manage printer queues, if every user wants to have a printer on their desktop?

This is solved using the local-<xterm> capability. One can print to any users X terminal/workstation, by sending the job to local-<hostname>. Since the queues and spooling directories are created dynamically, the administrators do not have to get involved in managing these queues. Any user/department can purchase a cheap inkjet printer and connect it to their terminals.

How does one set up printer lists in each application, so that users can print to any one of the hundreds of printers available?

Each application is configured to only print to "myprinter". This queue is automatically routed to the users user-<username> queue. Instead of having the printer manage the list of applications, this functionality is moved into the printing system (more details in "Local Queues" section).

How does one get print jobs from non-Unix systems to printer to their printers, with cover pages with Unix login names, and proper accounting, without much configuration on the Unix side?

The print jobs from non-Unix systems are received on a special queue that does non-Unix username to Unix username mapping. The job can then be

routed to "myprinter", and follow the usual print path.

There is only need for one queue on the non-Unix system, which will be used to send jobs to all possible printers on the Unix system. (More details in "Other systems and Legacy Printing".)

How does one distribute printer configuration (aka /etc/printcap) information to all their workstations/XDM servers?

The only printer configuration information stored on the clients is the hostname of the print spoolers, which is the same for all clients, and never changes.

The clients, regardless of queue name, always send the jobs to a print spooler. The solution is to move all the spooling and configuration information off of the clients, and put them on the "print spooler" hosts.

Can one build a print system so that if any of the print servers go down, all printing service can continue?

In this implementation, there are two PS servers, and two PD servers. In theory it is possible to have any arbitrary number.

If a PS server were to fail; clients would connect another PS server using the round robin multiple A record in DNS.

If a PD server were to fail; PD servers will use the next available server defined in the server list for that queue.

Therefore, if any server fails, the print service is still available for any new job. The only loss would be jobs that are in queue on the server that went down.

Discussion

Roll out

The roll out of the new print system was not without its problems.

By the time this project was started, the entire system administrator team had changed. Having noone know how the old system was set up in all its intricate details was a big hindrance during the roll out. It seems, no matter how hard the team tried, there was always, yet another undocumented feature, or an application that used the old system with hard-coded entry points. Parts of the old system that did not initally make sense to the team, seemed crystal clear once the same modes of failure were encountered in the new system (e.g., WordPerfect assuming that hosts were always single user, and see the print system fail in mysterious ways on multiuser XDM servers.)

Another problem was keeping up-to-date with the new versions of LPRng. During the development cycle of the project, there was a new version of LPRng almost every other week, and it contained changes that the project team had contributed or changes that were desperately needed. Of course, there were several occasions where LPRng had changed in a way that had not anticipated, especially in the bleeding edge areas that this system depended on most: control file filters and routing filters.

Failsafe

As stated earlier in this paper, the failsafe requirement of this project only applied to new jobs being submitted to the system.

It would have been a pretty difficult design challenge if this requirement also applied to existing jobs. After all, if a print job is currently on server A, and server A goes down; how does one recover the print job? One would have had to build a system that tracked print jobs on multiple servers simultaneously: not a simple feat!

One side effect of the system as implemented is that queue listings maybe inconsistent. The queue listing requests (lpq) go through the same failsafe mechanisms as the jobs. As a result, if one submits a job, and queries the queue immediately, he may not see his job: His job may be being processed on a different PS server than the one he is using to get the queue listing.

Another interesting side effect is the disappearance of jobs when a PD server is resurrected. While a PD server is down, print jobs that would have been processed on it are sent to a different server. When the PD is back up, the system will automatically switch back to using the PD server for queue listings. However, there may be jobs queued on a different PD server waiting to be printed.

Weaknesses In Implementation

The dynamic printcap generation script, /etc/printcap.pl is currently implemented in perl. This means that a new copy of perl is spawned every time lpd needs some printcap information. Under heavy printing, this can put a nontrivial load on the system.

The PS servers utilize users' preferences when processing print jobs. Currently these preferences are stored on a flat file on a NFS mounted filesystem. The design was to store these files locally on all PS servers, and when the user updated their preferences, update all copies.

Another weakness is in the implementation of Zephyr. Since the environment had no prior installation of Zephyr, it was installed specifically for this project. During the implementation Zephyr was used, without much regard to how else it could be used. Therefore, to use the current installation of Zephyr as a general purpose messaging mechanism may involve some rework in the print system implementation.

Comparison To Prior Work

In [8], the author uses a number of sections to cover printers, and their configurations. As an observant reader will note, this paper does not cover the configurations (and difficulties associated with) the printers themselves. One interesting feature covered in his paper; that can also be applied to this work in the

future, is the assignment of a location to print servers. In the current implementation of this print system, all the PS servers are treated equally. In a future version, it is possible to assign locations to these servers, and have clients contact a print spooler close to their location. The author also has several sections on the importance of automation. In this system, the approach taken was to "eliminate" instead of automating a lot of the maintenance work. For example, instead of distributing printer information to all the clients, the system was designed such that the client configuration is not affected by changes to the print system; and have, in effect, eliminated the need to distribute this information.

In [9], the authors describe a print system that they have built using the LPRng package. In their paper, they describe a design that they rejected that is similar in nature to this system: "in which a master front end machine distributes print jobs to an array of workers" because they considered the front end machine a single point of failure. This system uses multiple front end machines to accomplish the distribution, and thus does not suffer from the single point of failure. Moreover, in this system, when a server failed all new jobs are automatically routed around the failure; as opposed to the CERN work, which in the authors words: "In case of a failure of one of the servers, a reconfiguration of the naming service databases suffices to reallocate the queues served by the failed server onto active ones." Another comment the authors make in their "Future Developments" section is that they would like to implement user notification via Zephyr, which is already implemented in this system.

It is unfortunate that the work described in this paper had already been completed by the time [8] and [9] were published.

Availability

This system is based on the LPRng print system, whose source is available from the LPRng web site [2]

Due to the legal requirements of the company where this work was performed, the additional sources (such as /etc/printcap.pl) can not be distributed.

Future Work

There are several possibilities for future work in this area:

- As suggested in [8], implement the idea of assigning locations to the PS servers. This should make the system scale much better.
- Consider using lbnamed [5] to load balance to PS servers.

Author Information

Giray Pultar received a Bachelor of Science degree in Engineering and a Bachelor of Arts degree in Economics from Swarthmore College in 1994. He worked as a System Administrator at Motorola in the Cellular Infrastructure Group in Arlington Heights, IL until 1996; followed by a similar position at Abbott Laboratories in Abbott Park, IL until 1998. He is currently working as a consultant at John Hancock Funds in Boston, MA. He can be reached at <giray@coubros.com>

References

[1] *LPRng – An Enhanced Printer Spooler System*, Patrick Powell, and Justin Mason, LISA '96.

[2] http://www.astart.com/LPRng .

[3] Man pages: *lpr*, *lpd*.

[4] VT100 terminal documentation – printer mode escape sequences.

[5] *lbnamed, Load Balancing Name Server.*

[6] HP Envizex X terminal documentation.

[7] NCD X NCD 17/19 documentation.

[8] *Building An Enterprise Printing System*, Ben Woodard, LISA '98.

[9] *Large Scale Print Spool Service*, Ignacio Reguero, David Foster, and Ivan Deloose, LISA '98.

Snort – Lightweight Intrusion Detection for Networks

Martin Roesch – Stanford Telecommunications, Inc.

ABSTRACT

Network intrusion detection systems (NIDS) are an important part of any network security architecture. They provide a layer of defense which monitors network traffic for predefined suspicious activity or patterns, and alert system administrators when potential hostile traffic is detected. Commercial NIDS have many differences, but Information Systems departments must face the commonalities that they share such as significant system footprint, complex deployment and high monetary cost. Snort was designed to address these issues.

Introduction

Snort fills an important "ecological niche" in the the realm of network security: a cross-platform, lightweight network intrusion detection tool that can be deployed to monitor small TCP/IP networks and detect a wide variety of suspicious network traffic as well as outright attacks. It can provide administrators with enough data to make informed decisions on the proper course of action in the face of suspicious activity. Snort can also be deployed rapidly to fill potential holes in a network's security coverage, such as when a new attack emerges and commercial security vendors are slow to release new attack recognition signatures. This paper discusses the background of Snort and its rules-based traffic collection engine, as well as new and different applications where it can be very useful as a part of an integrated network security infrastructure.

Snort is a tool for small, lightly utilized networks. Snort is useful when it is not cost efficient to deploy commercial NIDS sensors. Modern commercial intrusion detection systems cost thousands of dollars at minimum, tens or even hundreds of thousands in extreme cases. Snort is available under the GNU General Public License [GNU89], and is free for use in any environment, making the employment of Snort as a network security system more of a network management and coordination issue than one of affordability.

What is "lightweight" intrusion detection?

A lightweight intrusion detection system can easily be deployed on most any node of a network, with minimal disruption to operations. Lightweight IDS' should be cross-platform, have a small system footprint, and be easily configured by system administrators who need to implement a specific security solution in a short amount of time. They can be any set of software tools which can be assembled and put into action in response to evolving security situations. Lightweight IDS' are small, powerful, and flexible enough to be used as permanent elements of the network security infrastructure.

Snort is well suited to fill these roles, weighing in at roughly 100 kilobytes in its compressed source distribution. On most modern architectures Snort takes only a few minutes to compile and put into place, and perhaps another ten minutes to configure and activate. Compare this with many commercial NIDS, which require dedicated platforms and user training to deploy in a meaningful way. Snort can be configured and left running for long periods of time without requiring monitoring or administrative maintenance, and can therefore also be utilized as an integral part of most network security infrastructures.

What is Snort?

Snort is a libpcap-based [PCAP94] packet sniffer and logger that can be used as a lightweight network intrusion detection system (NIDS). It features rules based logging to perform content pattern matching and detect a variety of attacks and probes, such as buffer overflows [ALE96], stealth port scans, CGI attacks, SMB probes, and much more. Snort has real-time alerting capability, with alerts being sent to syslog, Server Message Block (SMB) "WinPopup" messages, or a separate "alert" file. Snort is configured using command line switches and optional Berkeley Packet Filter [BPF93] commands. The detection engine is programmed using a simple language that describes per packet tests and actions. Ease of use simplifies and expedites the development of new exploit detection rules. For example, when the IIS Showcode [IISBT99] web exploits were revealed on the Bugtraq mailing list [BTQ99], Snort rules to detect the probes were available within a few hours.

Snort vs. The World!

Snort shares commonalities with both sniffers and NIDS. Two programs that lend themselves to direct comparison with Snort, tcpdump and Network Flight Recorder [NFR97], will be examined and contrasted in this section. In many cases, Snort is financially, technically, and/or administratively easier to implement than other Open Source [OSS98] or commercially available tools.

How Is Snort Different From tcpdump?

Snort is cosmetically similar to tcpdump [TCPD91] but is more focused on the security applications of packet sniffing. The major feature that Snort has which tcpdump does not is packet payload inspection. Snort decodes the application layer of a packet and can be given rules to collect traffic that has specific data contained within its application layer. This allows Snort to detect many types of hostile activity, including buffer overflows, CGI scans, or any other data in the packet payload that can be characterized in a unique detection fingerprint.

Another Snort advantage is that its decoded output display is somewhat more user friendly than tcpdump's output. Snort does not currently lookup host names or port names while running, which is a function that tcpdump can perform. Snort is focused on collecting packets as quickly as possible and processing them in the Snort detection engine. Performing run-time host name lookup is not conducive to high performance packet analysis. Figure 1 shows typical Snort output for a telnet banner display, and Figure 2 shows the same packet as displayed by tcpdump.

One powerful feature that Snort and tcpdump share, is the capability to filter traffic with Berkeley Packet Filter (BPF) commands. This allows traffic to be collected based upon a variety of specific packet fields. For example, both tools may be instructed via BPF commands to process TCP traffic only. While tcpdump would collect all TCP traffic, Snort can utilize its flexible rules set to perform additional functions, such as searching out and recording only those packets that have their TCP flags set a particular way or containing web requests that amount to CGI vulnerability probes. The SHADOW IDS [SHD98] from the Naval Surface Warfare Center is based on tcpdump and uses extensive BPF filtering. SHADOW is discussed in more detail near the end of this paper.

Snort and NFR

Perhaps the best comparison of Snort to NFR is the analogy of Snort as little brother to NFR's college-bound football hero. Snort shares some of the same concepts of functionality as NFR, but NFR is a more flexible and complete network analysis tool. That said, the little brother idea could be extended in that Snort tends to fit into small places and is somewhat more "nimble" than NFR. For example, NFR's packet filtering n-code language is a serious, full featured scripting language, while Snort's rules are more one dimensional. On the other hand, writing a Snort rule to detect a new attack takes only minutes once the attack signature has been determined. See Appendix A for an example of a simple web detection rule written in n-code and the analogous Snort rule.

NFR also has a more complete feature set than Snort, including IP fragmentation reassembly and TCP stream decoding. These features are essential in any commercial product that is meant to perform mission critical intrusion detection, and NFR was the first product which could defeat anti-NIDS attacks outlined by Ptacek and Newsham [PTA98]. Presently, Snort does not implement TCP stream reassembly, but future versions will implement this capability. Snort currently addresses IP fragmentation with a rule option that sets a minimum size threshold for fragmented packets. This rule option takes advantage of

```
20:59:49.153313 0:10:4B:D:A9:66 -> 0:60:97:7:C2:8E type:0x800 len:0x7D
192.168.1.3:23 -> 192.168.1.4:1031 TCP TTL:64 TOS:0x10 DF
***PA* Seq: 0xDF4A6536   Ack: 0xB3A6FD01   Win: 0x446A
FF FA 22 03 03 E2 03 04 82 0F 07 E2 1C 08 82 04   ..".............
09 C2 1A 0A 82 7F 0B 82 15 0F 82 11 10 82 13 FF   ................
F0 0D 0A 46 72 65 65 42 53 44 20 28 65 6C 72 69   ...FreeBSD (elri
63 2E 68 6F 6D 65 2E 6E 65 74 29 20 28 74 74 79   c.home.net) (tty
70 30 29 0D 0A 0D 0A                              p0)....
```

Figure 1: Typical Snort telnet packet display.

```
20:59:49.153313 0:10:4b:d:a9:66 0:60:97:7:c2:8e 0800 125: 192.168.1.3.23 >
192.168.1.4.1031: P 76:147(71) ack 194 win 17514 (DF) [tos 0x10] (ttl 64,
id 660)
                        4510 006f 0294 4000 4006 b48d c0a8 0103
                        c0a8 0104 0017 0407 df4a 6536 b3a6 fd01
                        5018 446a d2ad 0000 fffa 2203 03e2 0304
                        820f 07e2 1c08 8204 09c2 1a0a 827f 0b82
                        150f 8211 1082 13ff f00d 0a46 7265 6542
                        5344 2028 656c 7269 632e 686f 6d65 2e6e
                        6574 2920 2874 7479 7030 290d 0a0d 0a
```

Figure 2: The same telnet packet as displayed by tcpdump.

the fact that there is virtually no commercial network equipment on the market that fragments packets smaller than 256-bytes. By setting this threshold value to some reasonable value, say 128-bytes, fragmented packet probes and attacks can be logged and alerts can be sent by Snort automatically. Full IP fragment and TCP stream reassembly and analysis will be addressed in later versions of Snort.

Under the Hood

Snort's architecture is focused on performance, simplicity, and flexibility. There are three primary subsystems that make up Snort: the packet decoder, the detection engine, and the logging and alerting subsystem. These subsystems ride on top of the libpcap promiscuous packet sniffing library, which provides a portable packet sniffing and filtering capability. Program configuration, rules parsing, and data structure generation takes place before the sniffer section is initialized, keeping the amount of per packet processing to the minimum required to achieve the base program functionality.

The Packet Decoder

The decode engine is organized around the layers of the protocol stack present in the supported data-link and TCP/IP protocol definitions. Each subroutine in the decoder imposes order on the packet data by overlaying data structures on the raw network traffic. These decoding routines are called in order through the protocol stack, from the data link layer up through the transport layer, finally ending at the application layer. Speed is emphasized in this section, and the majority of the functionality of the decoder consists of setting pointers into the packet data for later analysis by the detection engine. Snort provides decoding capabilities for Ethernet, SLIP, and raw (PPP) data-link protocols. ATM support is under development.

The Detection Engine

Snort maintains its detection rules in a two dimensional linked list of what are termed Chain Headers and Chain Options. These are lists of rules that have been condensed down to a list of common attributes in the Chain Headers, with the detection modifier options contained in the Chain Options. For example, if forty five CGI-BIN probe detection rules are specified in a given Snort detection library file, they generally all share common source and destination IP addresses and ports. To speed the detection processing, these commonalities are condensed into a single Chain Header and then individual detection signatures are kept in Chain Option structures.

These rule chains are searched recursively for each packet in both directions. The detection engine checks only those chain options which have been set by the rules parser at run-time. The first rule that matches a decoded packet in the detection engine triggers the action specified in the rule definition and returns.

A major overhaul of the detection engine is currently in the planning and development stage. The next version of the engine will include the capability for users to write and distribute plug-in modules and

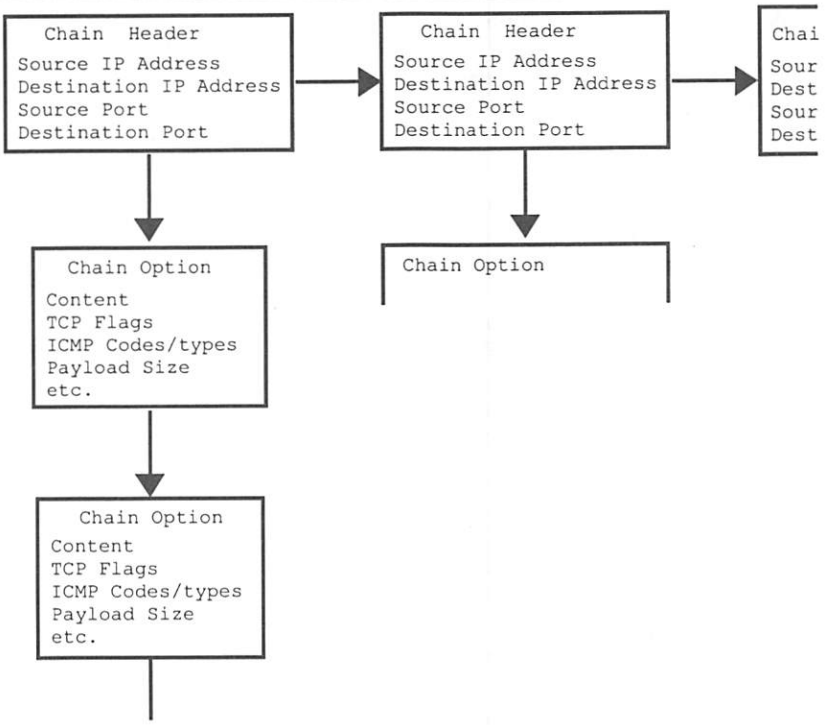


Figure 3: Rule Chain logical structure.

bind them to keywords for the detection engine rules language. This will allow anyone with an appropriate plug-in module to add significant detection functionality to Snort and customize the program for specific jobs.

The Logging/Alerting Subsystem

The alerting and logging subsystem is selected at run-time with command line switches. There are currently three logging and five alerting options. The logging options can be set to log packets in their decoded, human readable format to an IP-based directory structure, or in tcpdump binary format to a single log file. The decoded format logging allows fast analysis of data collected by the system. The tcpdump format is much faster to record to the disk and should be used in instances where high performance is required. Logging can also be turned off completely, leaving alerts enabled for even greater performance improvements.

Alerts may either be sent to syslog, logged to an alert text file in two different formats, or sent as WinPopup messages using the Samba smbclient program. The syslog alerts are sent as security/authorization messages that are easily monitored with tools such as swatch [SWT93]. WinPopup alerts allow event notifications to be sent to a user-specified list of Microsoft Windows consoles running the WinPopup software. There are two options for sending the alerts to a plain text file; full and fast alerting. Full alerting writes the alert message and the packet header information through the transport layer protocol. The fast alert option writes a condensed subset of the header information to the alert file, allowing greater performance under load than full mode. There is a fifth option to completely disable alerting, which is useful when alerting is unnecessary or inappropriate, such as when network penetrations tests are being performed.

Writing Snort Rules

Snort rules are simple to write, yet powerful enough to detect a wide variety of hostile or merely suspicious network traffic. There are three base action directives that Snort can use when a packet matches a specified rule pattern: pass, log, or alert. Pass rules simply drop the packet. Log rules write the full packet to the logging routine that was user selected at run-time. Alert rules generate an event notification using the method specified by the user at the command line, and then log the full packet using the selected logging mechanism to enable later analysis.

The most basic rules contain only protocol, direction, and the port of interest, such as in Figure 4.

```
log tcp any any -> 10.1.1.0/24 79
```

Figure 4: A simple Snort rule.

This rule would record all traffic inbound for port 79 (finger) going to the 10.1.1 class C network address space.

Snort interprets keywords enclosed in parentheses as "option fields". Option fields are available for all rule types and may be used to generate complex behaviors from the program, such as in Figure 5.

The rule in Figure 5 would detect attempts to access the PHF service on any of the local network's web servers. If such a packet is detected on the network, an event notification alert is generated and then the entire packet is logged via the logging mechanism selected at run-time.

The rule IP address and port specifiers have several features available. The CIDR block netmask may be set to any value between one and thirty-two. Port ranges can be specified using the colon ":" modifier. For example, to monitor all ports upon which the X Windows service may run (generally 6000 through 6010), the port range could be specified with the colon modifier as shown in Figure 6.

Both ports and IP addresses can be modified to match by exception with the bang "!" operator, which would be useful in the rule described in Figure 7 to detect X Windows traffic from sources outside of the network.

This rule would generate an alert for all traffic originating outside of the host network that was bound for internal X Windows service ports.

Snort version 1.2.1 has fourteen option fields available:

1. content: Search the packet payload for the a specified pattern.
2. flags: Test the TCP flags for specified settings.
3. ttl: Check the IP header's time-to-live (TTL) field.

```
alert tcp any any -> 10.1.1.0/24 80 (content: "/cgi-bin/phf"; msg: "PHF
probe!";)
```

Figure 5: Options allow increased rule complexity.

```
alert tcp any any -> 10.1.1.0/24 6000:6010 (msg: "X traffic";)
```

Figure 6: An example of port ranges.

```
alert tcp !10.1.1.0/24 any -> 10.1.1.0/24 6000:6010 (msg: "X traffic";)
```

Figure 7: Matching by exception on the source IP address

4. itype: Match on the ICMP type field.
5. icode: Match on the ICMP code field.
6. minfrag: Set the threshold value for IP fragment size.
7. id: Test the IP header for the specified value.
8. ack: Look for a specific TCP header acknowledgement number.
9. seq: Log for a specific TCP header sequence number.
10. logto: Log packets matching the rule to the specified filename.
11. dsize: Match on the size of the packet payload.
12. offset: Modifier for the content option, sets the offset into the packet payload to begin the content search.
13. depth: Modifier for the content option, sets the number of bytes from the start position to search through.
14. msg: Sets the message to be sent when a packet generates an event.

These options may be combined in any manner to detect and classify packets of interest. The rule options are processed using a logical AND between them; all of the testing options in a rule must be true in order for the rule to generate a "found" response and have the program perform the rule action.

Rule Development

Snort is extremely useful for rapidly developing new Snort rules. The clear and concise manner in which the data is displayed by the tool makes it perfect for writing new rules. The general method for development consists of getting the exploit of interest, such as a new buffer overflow, running the exploit on a test network with Snort recording all traffic between the target and attack hosts, and then analyzing the data for a unique signature and condensing that signature into a rule. Figure 8 shows Snort's view of a notional "IMAP buffer overflow" that has just come into widespread use by the "script kiddie" community.

The unique signature data in the application layer is the machine code just prior to the /bin/sh text string, as well as the string itself. Using this information, a new rule can be developed quickly, such as the one defined in Figure 9.

The content field of the rule contains mixed pain text and hex formatted bytecode, which is enclosed in pipes. At run-time, this data is converted into its binary representation, as displayed in the decoded packet dump in Figure 8, and then stored in an internal list of rules by Snort. Thus, the rule contained in Figure 9 will raise an alarm any time a packet containing the "fingerprint" of the new IMAP buffer overflow is detected.

Writing High Performance Pattern Matching Rules

The current rules system lends itself to high performance under most conditions, but there are some general concepts that can be applied when writing Snort rules to keep the processing speeds as high as possible. Computationally, the content matching option is the most expensive process that can be performed in the detection engine. Accordingly, it is performed after all other rule tests. This fact can be used to advantage by specifying other rule options in combination with the content option. For example, almost all requests to web servers have their TCP PUSH and ACK flags set. Using this knowledge, it is relatively easy to write a rule which will perform a simple TCP flag test before running the far more computationally intensive pattern match test.

Other options can be combined with the content rules to limit the amount of data that must be searched. The offset and depth keywords were made specifically to fulfill this function. Using these options, the area of the packet payload to search for an exploit pattern can be localized. Care should be taken to avoid limiting the search too severely. For example, many buffer overflows use variable offsets to tune the size and placement of the exploit machine code. A Snort rule that has been tuned too tightly to key on a specific area of a packet's payload may overlook the real exploit that has been shifted to a different area within

```
052499-22:27:58.403313 192.168.1.4:1034 -> 192.168.1.3:143
TCP TTL:64 TOS:0x0 DF
***PA* Seq: 0x5295B44E   Ack: 0x1B4F8970   Win: 0x7D78
90 90 90 90 90 90 90 90 90 90 90 90 90 EB 3B  ..............;
5E 89 76 08 31 ED 31 C9 31 C0 88 6E 07 89 6E 0C  ^.v.1.1.1..n..n.
B0 0B 89 F3 8D 6E 08 89 E9 8D 6E 0C 89 EA CD 80  .....n....n.....
31 DB 89 D8 40 CD 80 90 90 90 90 90 90 90 90 90  1...@...........
90 90 90 90 90 90 90 90 90 90 90 E8 C0 FF FF FF  ................
2F 62 69 6E 2F 73 68 90 90 90 90 90 90 90 90 90  /bin/sh.........
```

Figure 8: Notional

```
alert tcp any any -> 192.168.1.0/24 143 (content:"|E8C0 FFFF FF|/bin/sh";
msg:"New IMAP Buffer Overflow detected!";)
```

Figure 9: Alert rule for the new buffer overflow.

the packet. On the other hand, web CGI probes and attacks generally all take place at the beginning of the packet within the first thirty to fifty bytes. This can be a great place to optimize Snort content searching.

The actual search pattern used in the content rule is another area where performance tuning may take place. Snort uses a Boyer-Moore [SEDG97] algorithm to perform its pattern matching, which is one of the best algorithms available for that task. It achieves its greatest efficiency in cases where the pattern to match consists of non-repeating sets of unique bytes. For example, the Intel x86 architecture uses the hex value 0x90 to indicate a NOP in machine code. Buffer overflows generally use large regions of NOPs to pad the actual exploit code and make the return jump calculations easier for the exploit programmer. When specifying content match patterns, it is best to avoid including any NOPs in the match pattern, which will otherwise cause the Boyer-Moore routine to complete many partial matches before actually finding the correct match pattern.

Advanced Snorting

Snort is a flexible tool with a wide variety of uses. It is intended to be used in the most classic sense of a network intrusion detection system. It examines network traffic against a set of rules, and alerts administrators to suspicious network activity so that they may react appropriately. There are many other areas where Snort can be useful as well.

Shoring Up Commercial IDS's

Snort can be used to fill holes in commercial vendor's network-based intrusion detection tools, such as when a new attack makes its debut in the hacker/cracker community and signature updates are slow to come from the vendor. In this case, Snort may be used to characterize the new attack by running it locally on a test network and determining it's signature. Once the signature is written into a snort rule, the BPF command line filtering may be used to limit the traffic that Snort analyzes to the service or protocol of interest. Snort can be used as a very specialized detector for a single attack or family of attacks in this mode.

The recent IRDP denial of service attack [IRDP99] revealed by the L0pht provides a good example of this concept. The same day that the attack was announced, Snort rules were made available by the user community and these attacks were detectable.

Passive Traps

Another application to which Snort is very well suited is as a Honeypot monitor. Honeypots are programs or computers that are dedicated to the notion of deceiving hostile parties interested in a network. Most honeypot systems, for example Fred Cohen & Associates Deception Toolkit [DTK98], record their data at the server level, with a fake "service", such as an FTP server actually recording the data sent to it. The problem with that concept is that the services doing the recording have to be started before they will record anything. This means that events such as stealth port scans or binary data streams will be missed or garbled on honeypots that don't perform packet level monitoring. Another problem is that the data generated by such a system will tend to be complex by its nature.

The data coming out of a honeypot requires a skilled analyst to properly interpret the results. Snort can be a great help to the analyst/administrator with its packet classification and automatic alerting functionality. With these capabilities a honeypot can be erected as a stand alone intrusion detection mechanism. It requires no other monitoring or maintenance because Snort can be set to record and generate event notification on the first packet that arrives at the honeypot.

Snort can be used to implement another concept that is being advocated today; that of "passive traps" [MJR99]. A passive trap uses the "home field advantage" that network administrators enjoy when securing their networks. One aspect of this concept is that administrators know which services are <B>not</B> available on their networks. Snort rules can be written that watch for traffic headed for these non-existent services. Packets which are found to be using these ports may be an indication of port scanning, backdoors, or other hostile traffic. For example, a network that is not using TFTP can be configured with Snort alert rules for all packets headed to or from any node on the network bound for port 69. This can be a good method for detecting covert communications channels such as Loki or backdoors like Back Orifice. Another easy concept to implement to set up pass rules for all of the services known to be running on a network and log inbound connections to other ports or port ranges.

Shining Some Light on SHADOW

SHADOW is designed to be a cheap alternative to commercial NIDS. As an aside, SHADOW was probably the first true lightweight intrusion detection system. tcpdump is used as the sensor in these systems, which are configured using often extensive BPF commands. All traffic that is not filtered out with these BPF rules is collected into a single file that can become quite large over extended periods of time. Once the data is collected by the sensor, it is post-processed using a variety of external third party tools. There are some limitations to this system, including a complete lack of real-time alerts and a lack of good data classification tools to aid the analyst in identifying the data produced by the sensor.

Snort uses the same BPF filter language rules as tcpdump, and can be used as a complete replacement for tcpdump sensors in environments where SHADOW is the IDS of choice. The advantages of using Snort as a replacement sensor include real-time automatic traffic classification as it is collected and real-time alerting. This allows security events to be detected and acted upon by the administrative staff in

a more timely manner and log file sizes to be reduced significantly. At the same time, Snort can record the data it collects to tcpdump formatted files so that the data generated by the system can be post-processed for in depth analysis with existing tools that analysts are comfortable using.

Focused Monitoring

"Focused monitoring" is the concept of watching a single critical node or service on a network for signs of hostile activity. For example, the Sendmail [ALMN99] SMTP server has an extensive and well known list of vulnerabilities and exploits. A single Snort sensor could be deployed with a rule set that covers all known Sendmail attacks and would provide highly focused monitoring of that specific traffic on the network. These rules could even be extended to provide a running narrative of all of the commands and responses into and out of SMTP servers on the defended network. This can make the network security analysts job somewhat easier by letting the collection engine (Snort) describe the normal flow of commands and responses as well as the attacks.

Focused monitoring can be especially useful in instances where existing NIDS provide inadequate coverage. For example, a set of rules that monitor SQL database queries to a web or database server could be developed. This would provide more complete coverage of CGI and ODBC SQL attacks and probes than any commercial NIDS on the market today. This concept can be extended to any network communications technology that is under represented by commercial NIDS.

Conclusions

Snort was designed to fulfill the requirements of a prototypical lightweight network intrusion detection system. It has become a small, flexible, and highly capable system that is in use around the world on both large and small networks. It has attained its initial design goals and is a fully capable alternative to commercial intrusion detection systems in places where it is cost inefficient to install full featured commercial systems.

Availability and Requirements

Snort will run on any platform where libpcap will run. The current version of Snort is 1.2.1, and libpcap is required to compile and run the software. Snort is known to run on RedHat Linux 5.1/5.2/6.0, Debian Linux, MkLinux, S/Linux, HP-UX, Solaris 2.5.1-2.7 (x86 and Sparc), x86 Free/Net/OpenBSD, M68k NetBSD, and MacOS X.

Information about snort may be acquired directly from the author's web site at http://www.clark.net/~roesch/security.html .

Snort may be downloaded from the author's web site at http://www.clark.net/~roesch/snort-1.2.1.tar.gz .

There is a slowly growing library of Snort rules available at http://www.clark.net/~roesch/snort-lib .

Acknowledgements

Snort originally used Mike Borella's ipgrab program as a development template and example for how to properly code libpcap programs and packet decoders. ipgrab can be found at http://www.borella.net. Mike's code is an excellent starting point for any libpcap-based project.

Ron Gula of Network Security Wizards http://www.securitywizards.com provided valuable advice on logging methodologies and some of the initial program logic, as well as contributing example rules to the system.

Ken Williams <jkw@frey.rapidnet.com> has been fantastically supportive throughout the development of Snort, providing encouragement and ideas for additional features as well as providing a friendly forum for the distribution of Snort.

The Snort user community has been especially enjoyable to work with, providing bug reports, ideas for new development directions, and new rules for the library since the program's initial release. Their support and enthusiasm has kept this a vital and growing collaborative project far past what I had imagined was possible!

References

[SHD98] *SHADOW*, Steven Northcutt et al., Naval Surface Warfare Center Dahlgren Laboratory, 1998, http://www.nswc.navy.mil/ISSEC/CID/ .

[TCPD91] *tcpdump*, Van Jacobson, Craig Leres and Steven McCanne, Lawrence Berkeley National Laboratory, 1991, http://www-nrg.ee.lbl.gov/ .

[PCAP94] *libpcap*, Van Jacobson, Craig Leres and Steven McCanne, Lawrence Berkeley National Laboratory, 1994, http://www-nrg.ee.lbl.gov/ .

[DTK98] *Deception Toolkit*, Fred Cohen & Associates, 1998, http://all.net/dtk/dtk.html .

[GNU89] *GNU General Public License*, Richard Stallman, 1989, http://www.gnu.org/copyleft/gpl.txt .

[BPF93] "The BSD Packet Filter: A New Architecture for User-level Packet Capture," Steven McCanne, Van Jacobson, *USENIX Technical Conference Proceedings*, 1993.

[ALE96] "Smashing the Stack for Fun and Profit," Aleph1, *Phrack #49*, 1996, http://www.phrack.com .

[BTQ99] Bugtraq Mailing List, archives and vulnerability data base are available at Security Focus, http://www.securityfocus.com .

[IISBT99] "NT IIS Showcode ASP Vulnerability," *Bugtraq ID #167*, Parcens/L0pht, May, 1999, http://www.securityfocus.com .

[OSS98] *The Cathedral and the Bazaar*, Eric S. Raymond, 1998, http://www.tuxedo.org/~esr/writings/cathedral-bazaar/ .

[FYD97] "The Art of Port Scanning," Fyodor, *Phrack #51*, 1997, http://www.insecure.org/nmap/p51-11.txt.

[SWT92] "Centralized System Monitoring With Swatch," Stephen E. Hansen and E. Todd Atkins, *USENIX Seventh Systems Administration Conference*, 1993, http://www.stanford.edu/~atkins/swatch/lisa93.html.

[SEDG97] *Algorithms in C: Fundamentals, Data Structures, Sorting, Searching,* Robert Sedgewick, Addison-Wesely Publishing Company, 1997.

[IRDP99] *L0pht Security Advisory*, Silicosis and Mudge, August 1999, http://www.l0pht.com/advisories/rdp.txt.

[ALMN99] *Sendmail*, Eric Allman, 1999 http://www.sendmail.com.

[PTA98] *Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection*, Thomas Ptacek and Timothy Newsham, Secure Networks Inc, 1998, http://www.nai.com/services/support/whitepapers/security/IDSpaper.pdf.

[MJR99] *Burglar Alarms for Detecting Intrusions*, Marcus Ranum, NFR Inc, 1999, http://www.blackhat.com/html/bh-usa-99/bh3-speakers.html.

Author Information

Martin Roesch is a Network Security Engineer with Stanford Telecommunications Inc. He holds a B.S. in Computer Engineering from Clarkson University. He has extensive experience with intrusion detection systems and has developed several systems professionally. He was a primary software engineer during the development of GTE Internetworking's Global Network Infrastructure IDS, and designed and developed GTE's new commercial honeypot/deception system "Sentinel". He is also a member of the Trinux Linux Security Toolkit distribution development team. Snort is his first Open Source Software project, and has been an excellent learning experience for him. Contact him at <roesch@clark.net>.

Appendix A

Sample NFR rule to detect web server CGI probes (n-code sample excerpted from the L0pht's NFR IDS Modules web page at http://www.l0pht.com/NFR).

```
badweb_schema = library_schema:new( 1, ["time", "int",
                                        "ip", "ip", "str"], scope());

# list of web servers to watch. List IP address of servers or a netmask
# that matches all. use 0.0.0.0:0.0.0.0 to match any server
da_web_servers = [ 0.0.0.0:0.0.0.0 ] ;
query_list = [ "/cgi-bin/nph-test-cgi?",
               "/cgi-bin/test-cgi?",
               "/cgi-bin/perl.exe?",
               "/cgi-bin/phf?"
                ] ;
filter bweb tcp ( client, dport: 80 )
{
        if (! ( tcp.connDst inside da_web_servers) )
                return;
        declare $blob inside tcp.connSym;
        if ($blob == null)
                $blob = tcp.blob;
        else
                $blob = cat ( $blob, tcp.blob );
        while (1 == 1) {
          $x = index( $blob, "\n" );
          if ($x < 0)           # break loop if no complete line yet
            break;
          $t=substr($blob,$x-1,1);      # look for cr at end of line
          if ($t == '\r')
            $t=substr($blob,0,$x-1);    # tear off line
          else
            $t=substr($blob,0,$x);
          $counter=0;
          foreach $y inside (query_list) {
            $z = index( $blob, $y );
            if ( $z >= 0) {
              $counter=1;
              # save the time, the connection hash, the client,
              # the server, and the command to a histogram
              record system.time, tcp.connHash, tcp.connSrc, tcp.connDst,
                                                        $t to badweb_hist;
            }
          }
          if ($counter)
            break;
        }
        # keep us from getting flooded if there is no newline in the data
        if (strlen($blob) > 4096)
                $blob = "";
        # save the blob for next pass
        $blob = substr($blob, $x + 1);
}
badweb_hist = recorder ("bin/histogram packages/test/badweb.cfg",
        "badweb_schema" );
```

Appendix B: Snort rules to detect the same web CGI probes.

```
alert tcp any any -> any 80 (msg:"CGI-nph-tst-cgi";
                content:"cgi-bin/nph-test-cgi?"; flags: PA;)
alert tcp any any -> any 80 (msg:"CGI-test-cgi";
                content:"cgi-bin/test-cgi?"; flags: PA;)
alert tcp any any -> any 80 (msg:"CGI-perl.exe";
                content:"cgi-bin/perl.exe?"; flags: PA;)
alert tcp any any -> any 80 (msg:"CGI-phf";
                content:"cgi-bin/phf?"; flags: PA;)
```

Internet Routing and DNS Voodoo in the Enterprise

D. Brian Larkins – Lucent Technologies

ABSTRACT

This paper describes the process used to transition from a legacy intranet to a modern Internet access architecture. During the period of the Lucent/AT&T/NCR tri-vestiture, much care was given to re-engineer and re-design Lucent's data networking infrastructure with a modern and flexible design. As a part of this re-structuring, the existing Internet access architecture was viewed as archaic and in desperate need of redesign. The legacy intranet was isolated from the Internet in many ways including separate root name servers, a complete lack of routing information to or from the Internet, and everything was passed through home-grown application-layer proxy software. To remedy this, a project was created to provide transparent proxyless access to Internet hosts and applications. This project entailed designing a routing architecture that provided connectivity, redundancy, and manageability. In addition to routing issues, Lucent's DNS infrastructure would also need a redesign to handle new responsibilities given to it.

Introduction

In mid-1996, new packet-filtering firewalls were deployed to replace the existing (and aging) application-layer firewalls which proxied web traffic as well as telnet and ftp. These new firewalls were deployed in conjunction with new web proxy-caching servers. This alleviated much of the older firewalls' load, and provided web access faster by an order of magnitude. The plan at that time was to phase in the proxyless firewalls by making big improvements in web access and adding in support for generic TCP services later. The initial deployment of these firewalls with web proxies was called Phase 1.

The goal of the Phase 2 deployment of the proxyless firewall architecture was to enable internal clients to access Internet based TCP services. In order to provide this type of access to internal users, three primary problems needed to be solved. First, a mechanism for provid ing routing information to and from the Internet consonant with our routing policy had to be determined. The most difficult technical problem of this entire endeavor was designing the routing architecture to be resilient and robust, but also to not mess up our stateful packet-filtering firewalls with asymmetric routing. Second, the DNS needed to be modified to allow internal hosts the ability to resolve Internet hosts, while still retaining policy-based controls over such things as outgoing email, name resolution of joint-ventures, acquisitions, etc. Thirdly, (and perhaps the most stressful of all) was developing an outbound access policy that appeased both the R&D communities and the corporate security folks. This document primarily focuses on the technical problems related to implementing a proxyless routing and DNS infrastructure and will leave in-depth discussions on policy to other brave souls.

Historical Background

Many of the difficulties with deploying what might otherwise have been a simple design result from being split off from a 100-year old company with 310,000 employees. At the time Lucent was spun-off from AT&T, there were the equivalent of at least five "intranets" connected together in a tenuous fashion at best. During the separation of these networks, a new internal backbone was created [Umali96]. Using the hierarchal properties of the OSPF routing protocol, business-unit specific networks, regional networks (Europe, Asia, etc.), and special-purpose networks (e.g., research) were all knitted together [Moy94]. Also at this time, the WWW was becoming more and more vital to business. The result of the massive growth in IP network usage and the churn caused by splitting one of the world's largest private networks was a little too much for the old R&D-based Internet gateways to handle [Umali97].

At the time, the only way to access the Internet was through email, a web-proxy, or through a custom application-layer proxy [Cheswick94]. The proxy software required a customized client or linking source to a custom library. On systems which were well maintained by able system administrators this wasn't too much of a burden, but for developers who had unique environments, re-compiling and/or porting was troublesome. For most PC users, Internet access beyond the web or email wasn't an option. Given the 120,000+ PC users within the company, this was inadequate connectivity for a data/telecommunications equipment manufacturer.

In early 1997, a project was initiated to evaluate, design, and deploy a mechanism for providing access to the Internet without the need for customized client software. It became clear early on that this would be a

difficult task. The existing intranet infrastructure was never designed for easy integration with the Internet. In many ways Lucent was "self-rooted".

The primary internal name servers were configured to be root name servers in order to make it easy to control DNS policy such as mail flow. This was a carry over from the AT&T days. Self-rooting the DNS made the transition from AT&T to Lucent much easier, but it made it more difficult to integrate with the Internet. If requests arrived at the internal root name servers for external hostnames, they would be dropped instead of being forwarded to the appropriate DNS authority.

Likewise, the OSPF backbone was the core of Lucent's intranet. If the backbone didn't have a route to the destination network, the packet was dropped. This was more fortuitous than the DNS case because the lack of a default route on the backbone allowed us to consider it's use as a possible solution to some of our Internet routing problems.

The two most difficult issues to deal with were organizational issues and security policy issues. The sheer size of Lucent becomes readily apparent with any attempt to change the fundamental underpinnings of it's data network. The number of organizations involved in the operations and engineering of the network is truly staggering. Currently there are at least four major separate groups supporting WAN engineering, two groups supporting DNS, and another two supporting firewall policy and engineering. Getting buy-in from all the right groups was critical for the project's success, but was a project in and of itself to find out who needed to be involved and coordinating communication between them.

The organizational issues created extra requirements that might not ordinarily have appeared with a homogeneous networking group. It became desirable to partition responsibilities around organizational boundaries instead of technical ones. The one benefit that this provided inherently was broad peer review. As a result many design flaws were found and eliminated early in the design process.

Besides organizational issues, security policy quickly rose to the list of hot topics. Lucent traditionally has aligned itself more with financial institution security rather than Silicon Valley start-up. On one hand there is the R&D community which would like a university-like environment, and the corporate security folks on the other, which would prefer retina-scans prior to Internet usage. Discussions continue to this day regarding the overall access policy for Internet usage.

Design Philosophy

In order to decompose the proxyless Internet project into meaningful and manageable tasks, we split the effort into three principal areas: routing, DNS, and security policy issues. This left us with two technical problems and one policy problem. With respect to routing and DNS, traditional engineering principals were used to further break down the project into tasks. To help guide the design process we adopted goals for both the routing and DNS problems which helped specify the ideal architecture. This section presents the guidelines that were generated for the routing architecture. The DNS issues are presented below.

Routing Design Philosophy

The philosophy behind the routing architecture is as below.

1. **A proxyless routing architecture should be as simple as possible, but no simpler.**

The less complexity that there is in a routing design, the less possibility for errors. These errors include configuration problems, troubleshooting difficulties, route pollution, route loops, etc. A design that meets all the requirements for a proxyless routing architecture should be as simple as possible to implement, but robust enough to meet all design criterion.

2. **A proxyless routing architecture should be fault tolerant.**

An Internet access outage at any single point should not cause significant loss of service for the corporation. Redundancy should exist in all designs to prevent significant outages. Failure of connections active at the time of a fault is acceptable. All connections initiated after a fault should proceed through an alternate path.

3. **A proxyless routing architecture should be dynamic.**

Fault detection should be transparent and automatic. Routing around a lost network egress point should happen quickly and without human intervention. The faults that should be detected are a any loss of path connectivity to the Internet from a backbone/Internet border router. A loss of any component in the path should trigger a new route to be advertised to the corporate backbone.

4. **A proxyless routing architecture should be symmetric.**

Internet destined traffic should route out a selected gateway point. The response traffic from the external server should return to the original exit point, to be correctly routed to the internal host. In addition, routes should be stable internally, to ensure that Internet bound traffic is routed through the same gateway when originated by an internal host. Path flapping is not a desired method of traffic flow, even in the interest of load balancing.

NOTE: This design goal could change with the ability to effectively share stateful firewall connection information. It will still be desirable for all Internet bound traffic to take a deterministic path, though this path may not be strictly symmetric.

5. A proxyless routing architecture should be secure.

Routes generated from Internet Service Providers are not trusted. All routing information derived from exterior route peers is suspect, and thus should not be allowed to significantly influence traffic patterns within the corporate network. We should not trust that our service providers will protect our internal route tables from being polluted. For example, we should protect against accepting internal routing information from the Internet. Additionally, filtering needs to prevent the leaking of internal information (either routing updates or data traffic itself) to the Internet.

Further, route distribution mechanisms themselves should not pose significant security risks to the internal data network. Router to router communications should be strictly limited and significant protocol filtering should occur to limit the risk of contamination. This is even more important when the flow of information crosses between exterior (untrusted) and interior (trusted) networks.

Problems to be Solved – Part I: Routing

In order to provide routes to the Internet, it is essential that the corporate backbone routers be aware of exit points to the Internet. By the same token it is essential that the Internet have routes by which the traffic can be returned to the Lucent internal network. Solving the problem of how to route internal packets to the Internet is separate from how to route Internet originating packets back to the corporate intranet. We'll break the following discussion into two parts; the first examining traffic originating from the intranet and destined for the Internet and the second covering the converse.

Internal Routing Issues

Traffic destined for the Internet must be routed through one of the corporate Internet firewalls. Within the Lucent data network, routing is hierarchal. End-users are attached to sub-areas which typically run OSPF within the sub-area. Areas (in the OSPF sense) are assigned based on geography in the case of our European and Asian regions, organization in the case of some business units, as well as history for much of the R&D community.

No matter which routing protocol is used within each sub-area, the routing information is distributed into OSPF at the backbone through an OSPF *Area Border Router* (ABR). The ABR is a member of both the sub-area's routing domain as well as the backbone's area [Khan97]. Using OSPF terminology, the backbone is known as *Area 0*. As a result of the route redistribution performed by the ABR, the Area 0 backbone contains all the routing information for the entire company.

The benefit of this is that we can assume that for any packet, a local router either knows itself the path it should take within the sub-area or will forward it "up" to the backbone. A direct consequence of this is that routing packets to the Internet can be reduced to having the Area 0 routers be "Internet aware". The remaining routers within the intranet already forward packets destined to unknown networks to the Area 0 routers.

There are three primary ways to provide route information to the backbone routers with respect to Internet reachability: default routing, partial Internet routing, and full Internet routing. Default routing entails the distribution of either static or dynamically sourced default routes into the OSPF backbone. Any traffic that the backbone doesn't have a known route for will be forwarded to the Internet. Partial and full Internet routing means obtaining a partial or complete list of all the routes announced on the Internet and distributing them to routers attached to the backbone. By having a complete list of routes, it is possible for packets to take the shortest path out and perform load balancing.

Border Gateway Protocol 4 – Your Friend

No matter which alternative is chosen, the de facto routing protocol used to communicate between *Autonomous Systems* (AS) is the *Border Gateway Protocol* (BGP) [Chandra97] [Halabi97]. Autonomous systems are the way that the Internet is broken up into organizationally separate networks. BGP is the glue that knits every corporation, organization, university, and service provider together to the Internet. BGP is an *exterior gateway protocol* (EGP) because it was designed specifically to deal with routing between distinct autonomous systems. BGP contains features which allow network administrators to carefully control the sending and receiving of routing information. Whereas OSPF and IGRP are *interior gateway protocols* (IGPs) and designed to distribute routing information within an autonomous system, BGP is a tool to implement an organizations' routing policy in addition to exchanging routes.

BGP has two flavors which are determined by the manner in which BGP is configured [Rekhter95b]. First, exterior BGP (EBGP) is used when two BGP peers are in differing AS's. EBGP is used when exchanging routing information between our routers and that of our ISP's. Alternatively, interior BGP (IBGP) is used when two BGP peers are within the same AS. The primary difference between the two is in the rules that are used to exchange routes so as to prevent route loops. Another important difference is that EBGP peers should be directly connected, while IBGP peers only need to be able to connect via TCP [Rekhter95a]. The ramifications of this will be apparent later.

A Return to the Internal Routing Problem

The principal behaviors that are desired in the proxyless architecture are fault-tolerance (i.e. a single ISP can fail without a significant loss in connectivity), and symmetry (traffic that leaves a specific gateway,

returns through that gateway). These constraints and the network topology during the design phase of the proxyless routing architecture will lead us to the selected design.

Organizationally, the Internet perimeter is engineered and maintained by a separate organization than the corporate backbone. Consequently, operational and maintenance issues at the time prohibited the use of BGP directly on the core backbone [Umali97]. In addition, the routers used at each of the seven corporate Internet gateways are only connected through the corporate backbone. I.e. there are no dedicated WAN links between the perimeter routers.

These constraints effectively rule out the use of either partial or full Internet routing as a solution to the internal routing problem. When packets bubble up to the Area 0 backbone, they will eventually get to a BGP- speaking perimeter router. If the router has a full routing view of the Internet, it might decide that another gateway is actually closer to the end destination. Since the backbone is unaware that the alternative path is better (it's only running OSPF, remember?), it will simply forward the packet back to the perimeter BGP router that it forwarded it to last time. Voilà, route loop. In case you're lost, the rule is that IBGP must be running on every router in between multiple AS exit points when attempting to do shortest path routing (as with full or partial Internet routing). Since we have a constraint that doesn't allow BGP on the backbone and another constraint that doesn't allow dedicated WAN links to connect the perimeter routers, we outsmarted ourselves right into a default based architecture.

Since IBGP doesn't require that peers be directly connected, we can run IBGP between each Internet gateway site. The only requirement for IBGP peering is that peers can establish a TCP session with one another. By using IBGP to peer with each border router, exit points can be agreed upon and the corresponding routes injected into the Lucent backbone in a deterministic fashion. Using IBGP between multiple peers does however mandate that there exist a full mesh, or IBGP connections between all peers. The reason for this requirement is to preserve a loop-free topology. The use of BGP route reflectors or confederations [Halabi97] can minimize the complexity of such a topology, thus allowing BGP information to easily be "tunnelled" over our native OSPF backbone.

This strategy also provides redundancy and symmetry. The Lucent BGP routers can each accept a default route from each ISP. We can then weight these routes to prefer a single entry/exit point and inject the route for it into the backbone. If a failure should occur with the primary exit point, the default will cease to be advertised into BGP, which will select the next highest weight default route and proceed to advertise that into OSPF (only one default at a time). This will be discussed in depth below. While we trade optimal routing for a primary/backup type solution, we're making the best of the network we have and a huge improvement over the existing configuration.

In addition to the general traffic destined to the Internet there are various servers (mail, DNS, web proxy) that are located at each gateway location. The desired behavior is that these various servers should always use the local firewall, instead of following a BGP generated default. This causes these servers to always use the Internet connection that is local, as opposed to routing high-volume concentrated services such as web access and mail out the single primary egress point.

This effect can be accomplished with policy routing, which is a way to alter the route traffic takes based on it's source address as opposed to traditional routing based on destination address. In the case of an outage it is possible to have proxy traffic be routed to another firewall exit point, but again may cause load problems on the backbone. The exact mechanism used to accomplish this will be discussed in detail below.

External Routing

When routing traffic back to the Lucent data network, there are issues of symmetry, route disclosure, route announcement restrictions, and provider address independence to consider. There are two primary alternatives to provide routes back into Lucent's internal network: address translation or full route disclosure.

Network address translation (NAT) provides an elegant solution to issues of symmetry and route disclosure. By mapping the internal hosts' IP address to a small pool of Internet addresses we can fix the connection path between remote Internet server and Lucent perimeter gateway. This allows us to avoid a plethora of asymmetric routing problems that can cause serious trouble to stateful packet filters.

Address translation enables us to control the return path of outbound traffic. NAT also allows us to advertise reachability information to the Internet, without disclosing information about the internal network's topology. By hiding the original source address, it is non-trivial to discover our use of address space, and the way our networks are configured. This enhances the level of security provided by our firewalls. An additional constraint of using a NAT-based solution is that the internal route a host takes to the gateway must be consistent. For example, if the internal network isn't configured appropriately it would be possible for two packets to exit through separate gateways. Each gateway would translate the source address to a different mapped address which to the server would look like coming from two different hosts.

When considering a NAT solution it is important to understand how a given implementation of address translation scales. Since Lucent has more than 200,000 hosts, a one-to-one mapping of IP addresses could

theoretically consume more than three full class B networks. A TCP or UDP connection can be uniquely identified by the 5-tuple of (*src addr*, *src port*, *dst addr*, *dst port*, *protocol*) [Stevens94]. Given this, it is possible to map both the source port and address for many simultaneous connections to a single translated address if the source port is used to uniquely identify the original source address and port. Overloading a single IP address for hiding multiple internal addresses is called *port address translation* (PAT). Using PAT to hide internal addresses involves ugly issues like trusting your firewall to correctly handle ICMP, well-behaved expiration of translation table entries, etc. Our past experiences with address translation brought considerable skepticism that NAT/ PAT schemes could scale well. Even more devious, troubleshooting complicated problems with NAT in the loop has been exceedingly difficult when using off-the-shelf firewall software.

On the other hand, a full announcement solution would require the external border routers to announce routes for all internal Lucent networks. Lucent has around 115 class B-sized blocks (/16's) that would all have to be announced. Announcing these addresses discloses slightly more information than using NAT. By freely distributing these routes, it becomes easier to determine our internal network topology, and we also disclose the true IP address of an internal machine that accesses an external server.

Full announcement can be very beneficial by removing the added overhead and complexity of NAT. Most packet-filtering firewalls operate with higher performance without NAT policies installed. This also addresses complications that NAT can cause with higher-level protocols that encode the source IP address at the application layer (e.g. FTP). Also, the handling of ICMP packets that may have a translated address in the payload, instead of the expected internal IP address can at times be problematic as well.

Whichever solution is used, some routing information will need to be distributed to the Internet. If a NAT solution is chosen, the address blocks used for the NAT pools must be announced to the Internet by someone. This can be done by either the ISP or ourselves. If we announce routes ourselves, BGP4 is required to peer with our ISP's. On the other hand, we can trade control for complex routing if our ISP's announce the NAT blocks for us.

The Routing Design

After comparing all the positives and negatives of each possible design, we chose to take a primary/backup solution. This means that all direct Internet traffic flows out a single exit point, but with a dynamic failover. In the case of a failure with the primary site, a backup gateway will come online automatically and transparently. We also elected to announce all of our address space to the Internet, eliminating the need for NAT.

There are four primary components to the routing design:

- To advertise routes for Lucent's networks to the Internet
- To accept routes to the Internet from our ISP's
- To select appropriate routes for outbound traffic
- To advertise selected routes into the OSPF backbone

Inbound Traffic Route Announcements

In order for traffic originating from Lucent networks to the Internet to be routed back correctly, the Internet must have routes for Lucent's networks. The preferred way to do this is to perform full route announcement for all of Lucent's networks to the Internet. This simplifies the firewall configurations considerably, and also bypasses some side effects caused by firewalls performing address translation.

To announce all of the Lucent routes to the Internet, it is necessary to register RIPE-181 or RPSL compliant objects with the *Internet Routing Registry* (IRR). The IRR is a collection of several distributed routing databases. In particular, Lucent needs to submit it's routing updates to the Route Arbiter Database (RADB) which is a principal U.S. routing registry. Many ISP's use the IRR to directly generate filters that control the propagation and distribution of routes.

In addition to the routing registries, it is also vital to provide direct information via an external routing protocol directly with an ISP's router. The only way to exchange routes with all of our ISP's is by using BGP. The BGP peering that takes place between the different autonomous systems (Lucent's AS and the ISP's AS) is via External BGP, or EBGP.

To ensure that routing is symmetric, Lucent's routes will be announced at several gateways. A BGP trick to prioritize routes from the announcing end is to add additional hops to less desirable routes. This is achieved traditionally by prepending extra copies of one's own AS number on all outgoing route announcements. Routers elsewhere on the Internet will get one route from each gateway, but some will have extra AS hops tacked onto the AS path information. Since these routes take a longer path to reach the same network, the primary return path will be preferred.

In the case of a primary gateway failure, Internet routers will simply use the next shortest path, even though it may contain extra AS path information.

External Outbound Route Acceptance

In order for internal traffic to be dynamically routed to the Internet, routes must be advertised from our ISP's to the Lucent backbone. It is necessary to accept a default route generated from each ISP to determine the best available default to inject into the OSPF backbone.

If the physical link should fail between the EBGP peers, the BGP session will disappear between the two routers. When the session is dropped between

peers the routing information will expire, typically in about 90 seconds. This causes the default route to be withdrawn, which may or may not affect the route selection process.

Internal Route Selection

After the default routes have been received by all Lucent BGP speaking routers, the best available entry/exit point must be selected. "Best available" is determined by a policy decision based on usage of our internal WAN links and concentration of users. Each candidate gateway is given a weight according to policy (e.g. the primary gateway will have the best weight, the first secondary the next best, etc.)

By configuring a BGP attribute called *local preference*, each gateways' default routes are assigned weights. As part of the BGP route selection process, local preference is used to determine which route is selected from the BGP tables and entered into the router's route table.

OSPF Default Route Origination

In order to propagate this information into the OSPF backbone, a default route must be distributed based on the results of the route selection process used by BGP. This means that the router which announces the default route to the corporate intranet must speak both BGP and OSPF. This originally caused some consternation with the backbone engineering teams and eventually led to the introduction of the *innie-outie router* (described below).

Routing Implementation

This section references Figure 1 and walks through each component specifying the tasks that should be performed there.

ISP Router (isp-rtr)

BGP processes run on both the Lucent external router and also on the ISP's border router. These two BGP neighbors peer, and due to the differing AS numbers, agree to speak EBGP. The external ISP router advertises, via EBGP, a default route (to the Internet) to the external Lucent router (ext-rtr).

The configuration of this router is the responsibility of our ISP and entails negotiation with their engineering staff for proper setup. Different ISP's are willing to support different configurations. Minor adjustments to the architecture may need to be made to support the requirements of each ISP's infrastructure.

External Router (ext-rtr)

The external router maintains an EBGP peering session with the ISP router and receives a default route from it. It also peers with the innie-outie router (io-rtr) as well. The peering between the innie-outie router and the external router is still BGP, but because both routers are within the same AS, they speak Internal BGP or IBGP.

The external router announces reachability information for all Lucent networks to the Internet. Route information is obtained dynamically from the io-rtr. Because the outbound announcements are being generated from dynamically updated information, a link failure forces a revocation of these routes, causing traffic destined for the internal network to be routed to the next highest preferred gateway.

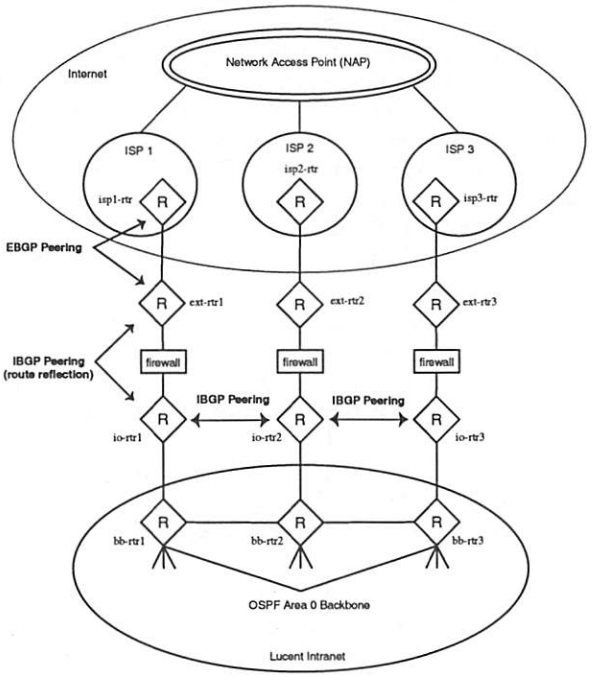


Figure 1: Network overview.

The AS path information is modified to prefer a return path through a particular entry/exit point [Halabi97]. When BGP routers on the Internet receive a packet destined to a Lucent network, they look in their BGP tables for all possible routes. If there are multiple routes in the table for the same network (each firewall location advertises routes for all Lucent networks), a comparison is done between all the entries to select the best route. The list of AS's that a route has traversed is called the AS-path. The shortest AS-path is considered the "best" route. By prepending multiple copies of Lucent's AS number, we can propagate multiple routes for the same network, but force traffic through a preferred location by extending the AS-paths of all the other routes. This mechanism allows us to weight preferred return paths back to the Lucent network.

It is important to note, that since routes are advertised and/or revoked by changes that affect the updating of Lucent routes to the external router, any link change may disrupt outgoing route announcements. When a route appears and disappears in rapid succession, this is called flapping. Since route flapping can bring the Internet to its knees, it is highly discouraged, and in most cases flapping routes will be

suppressed until they are stable. This could cause a serious service outage, and is to be avoided. This implies extreme care in taking down links or hosts that lie between the ext-rtr and the io-rtr.

To further reduce the risk of flapping routes, the external router will only announce aggregate (or "supernet") routes to the Internet [Fuller93]. This prevents the announcement of superfluous routing information and improves stability.

Firewall

Currently, the existing firewalls are stateful packet-filtering firewalls that act as Layer 2 networking devices (i.e., bridges). This may not always be true, and we want to retain the ability to choose firewalls that operate as Layer 3 networking devices (i.e., routers). Since the firewall component of this architecture may change, it is critical to be able to support either configuration in the routing architecture.

By using IBGP to exchange routing information between the io-rtr and the ext-rtr, there is no need to use a routing protocol on the firewall itself. IBGP allows a peering session to be set up between two routers that are not directly connected, but have IP reachability. If we had to use a routing protocol that mandated peers to be directly connected, the firewall would be required to run a routing protocol and peer with both external and internal routers. This could cause security problems and is not a good security practice.

Although the firewall is not running a routing protocol, it is vital that the firewall (if implemented as a router) have the ability to route information either to the Internet, or to the internal network. This is most easily accomplished by setting the default route on the firewall to the ext-rtr, and installing static routes for all internal networks into the routing table. This is quite a sizeable route table with a network the size of Lucent which leaves us predisposed to bridging firewalls. There is no need for any of the routing hassles on the firewall if it is implemented as a bridge [Limoncelli99].

A direct consequence of advertising all Lucent networks to the Internet is that Network Address Translation does not need to be configured on the firewall, avoiding complex firewall configurations and the woes that accompany them.

Innie-Outie Router (io-rtr or I/O router)

The io-rtr is responsible for several roles in the architecture. It accepts the default route from it's corresponding ext-rtr, and installs it into the BGP table. The BGP process on the io-rtr selects the preferred default route out, and injects that route into the OSPF backbone [Halabi95].

In order to exchange routing information between the ext-rtr and the io-rtr, an IBGP peering session must be established. Recall that IBGP has some requirements on its configuration to prevent route loops and other network inconsistencies. IBGP peers must be configured in a full mesh, but they do not have to be directly connected. When there is no direct connection between peers, it is essential that a route exists within the route table to reach the IBGP peer.

The problem with IBGP peering is that it mandates a full mesh between peers. This would have TCP sessions established between external routers and all other IBGP routers at other locations. This configuration is less than desirable.

By running the io-rtr as a BGP route reflector, it enables a single IBGP session between the external IBGP speaker and the innie-outie router, while the io-rtr's are fully meshed between themselves. [Chandra96] Traditionally, route-reflectors are used in situations where meshing is not feasible due to the number of BGP speakers, but it solves an otherwise untenable problem for this architecture. Route reflection in this application is akin to having one router proxy the IBGP mesh to the hard-to-reach external routers.

Once an IBGP session has been established, the ext-rtr and the io-rtr can exchange routes. Since the ext-rtr is not part of the trusted internal network and the io-rtr is behind the firewall, the io-rtr needs to be configured to accept only a default route from the external Lucent router. This enhances security by providing another layer of route filtering [Raza97].

Route distribution filters can be applied to updates sent or received from specific peers. By employing route filters on received routes, it's trivial to determine the origin of a route (i.e., the ext-rtr, or an io-rtr at another location). The origin can be used as a selector which weights the route to match traffic exit policy (by setting or comparing the local-preference attribute). The I/O router then selects the default route with the highest local-preference and installs it into the routing table. This default route is then injected into the OSPF backbone by the I/O router that is receiving the highest weighted default from its corresponding ext-rtr. At any one time, only one default route is ever injected into the OSPF backbone. This selection process is the core of the primary/backup architecture. The setting of local-preference values strictly determines the primary gateway and also the order in which backup gateways come online in the event of a failure. Proper configuration of the filters and local-preference manipulation is crucial to the correct operation of the architecture.

In order to route traffic back in, the io-rtr must be member of the OSPF Area 0 as an OSPF *Autonomous System Border Router* (ASBR). This allows OSPF to provide a complete list of all internal routes without having to statically define any routes on the io-rtr. These routes must be redistributed into BGP, but only for the benefit of the ext-rtr. Each io-rtr will receive all Lucent routes from a local OSPF neighbor, but will not exchange OSPF-originated routes via BGP to

other io-rtr's. This prevents any route loops or other sub-optimal routes from being generated [Rekhter94] [Varadhan92]. The IBGP configuration on the io-rtr will allow the ext-rtr to receive routing updates about Lucent networks. If there is a link outage between the io-rtr and ext-rtr, the ext-rtr stops advertising routes (after they timeout without an update) and the Internet will route traffic back to an alternate firewall location.

There are also web proxy servers, mail servers, and various other Internet servers that are connected to another interface on the io-rtr, which require that traffic always default out the local firewall. By following the BGP selected defaults, all web proxy traffic would be routed out a single firewall. In order to prevent this, policy routing can be used to force all web proxy traffic out the local firewall. Policy routing is used to alter the next hop address based on the packets source IP address (as opposed to classic routing, which determines next hop based on destination address). Any packet originating from these subnets will default route out the local exit point.

Although policy routing handles traffic from these servers to the Internet, it does not force proxy traffic back through the firewall that it went out of. The two ways to ensure that this happens are to advertise the servers' network from only a single gateway, or to perform address translation on the outbound proxy traffic. In order to be consistent with other routing policies, it is preferred to announce routes for networks that will normally be routed out the local gateway, instead of relying on NAT.

DNS Design Philosophy

The philosophy behind the DNS architecture is as below.

1. A proxyless DNS architecture should be as simple as possible, but no simpler.

Similar to the routing issues, as complexity increases so does the likelihood for errors. Again, a design that meets all the requirements for a proxyless routing architecture should be as simple as possible to implement, but robust enough to meet all design criterion.

2. A proxyless DNS architecture should be fault tolerant.

This requirement is also similar to the corresponding routing design goal. Internet access outage at any single point should not cause a significant loss of service for the corporation. Redundancy should exist in all designs to prevent significant outages. Failure of DNS resolution requests pending at the time of a fault is acceptable. All post-fault name lookup requests should automatically proceed to an alternate server.

3. A proxyless DNS architecture should be consistent.

When internal hosts resolve addresses, reverse lookups on that IP address should yield the corresponding name. If masquerading or NAT techniques are employed within any part of the proxyless design, the DNS should provide a consistent view of the network. Additionally, any controls for policy management of e-mail, joint ventures, mergers, or acquisitions should be similarly consistent.

4. A proxyless DNS architecture should handle policy based management of name resolution.

The Lucent/AT&T/NCR tri-vestiture signaled a new era of intranet churn in which the network should be resilient to faults, but adaptable to change. With mergers, acquisitions, and joint ventures happening on a monthly basis, it is critical that any new DNS infrastructure be able to support name resolution for non-Lucent business partners or recent acquisitions as well as Internet name resolution.

This includes the ability for internal hosts to send and receive email through a policy defined mechanism (i.e., through the internal link, or via the Internet). Also provisions for accessing both internally accessible partner sites as well as their Internet sites.

5. A proxyless DNS architecture should be secure.

While it is a necessity to be able to resolve Internet names and addresses, it is not desirable to release internal topology and structural information to the Internet. For example, it's desirable to access *ftp.abc.test*, but it's probably not wise to reveal the internal host as *5ess.source.lucent.com*.

In addition to not releasing internal host and domain names, it is also a risk to accept and trust information from Internet based DNS sources. As direct consequence, care should be taken when obtaining such information, as well as distributing the information internally without validation or screening of some sort.

Problems to be Solved – Part II: DNS

In order for any internal DNS name servers to provide answers to internal queriers, these servers must be able to send and receive queries to Internet name servers. Similar to setting up an intranet, routeability is a precursor to name resolution. For the sake of this discussion we'll presume that sufficient routing exists to facilitate whatever architecture is most appropriate.

The principal problem is convincing our internal top-level name servers (which you may remember are self-rooted) to forward queries for unknown domains to the Internet. At the same time, it extremely important to maintain strict control over both external and internal views of our DNS, as well as using the DNS to control email routing policies. Besides simply providing this functionality, these problems should be addressed in a way which doesn't add any significant additional security risks.

There are many well-known risks of blindly trusting information distributed to the public DNS

[Bellovin95]. Specifically, we want to avoid problems with contamination. Although there are several methods to do this, we were predisposed to some researchware called *dnsproxy* [Cheswick96]. Dnsproxy handles many of the policy and security issues in a unique and elegant way. The specific usage of dnsproxy is described in depth below.

The view we present the Internet of Lucent's internal network also must be carefully designed. Whether NAT or full route announcement is used, the IP addresses that appear on the Internet should resolve to valid hostnames. While the hostnames which are visible to Internet hosts should be valid, there is no requirement that they match the internal hostnames that the IP addresses correspond to. In fact, it is in the interest of security that they should not match the internal address.

The DNS Design

To simplify the DNS design, we decided to not reinvent the wheel and instead capitalize on a design solution from our compatriots in Bell Labs'. There are two principal problems to be solved with DNS, internal name resolution, and external reverse name resolution.

First, the internal DNS needs to be able to resolve Internet hosts. We chose to accomplish this by configuring the pre-existing root nameserver to forward unknown requests to a dnsproxy server. Dnsproxy acts as a switch and filter for DNS requests. Given a query for any resource record type it can switch the set of name servers to query for the correct answer. In addition to it's switching capabilities, it also provides filtering to protect internal queriers from DNS mischief. The dnsproxy source is approximately 4,000 lines of C code. More information on dnsproxy can be found in [Cheswick96].

The typical dnsproxy configuration from research involved determining whether the query should be routed internally, or to Internet name servers. For general corporate use we required some more complicated configuration and eventually further customized the server software. The many joint ventures, mergers, acquisitions, and spinoffs each require specialized DNS handling. Specifically, the treatment of MX records by name servers can get especially insane.

Our default policy is that the corporate email gateways handle all outgoing email. Beyond this, each back-door network connection should have mail either handled over the Internet or through the internal network, depending on the legal agreement.

Custom software again comes to the rescue to help solve the external reverse lookup problems. As stated above, we want to prevent the announcement of the *5ess.source.lucent.com* name to the Internet, but we still want to have functional DNS entries for every host. The way we chose to do this was by creating a new domain in the external DNS, *outland.lucent.com*. Any host that was not explicitly in the external DNS would be given an entry in the *outland.lucent.com* domain. Normally this would be done by creating PTR and A records for each possible internal IP address that pointed to a made-up name. We use the following format: for host 10.2.3.4, the PTR record points to h10-2-3-4.outland.lucent.com, and the A record for h10-2-3-4.outland.lucent.com is 10.2.3.4.

Unfortunately, creating all these records involves maintaining a nameserver with nearly 10 million entries (for all of Lucent's address space). To avoid having to deal with this, it was simpler to write a small piece of software to generate answers dynamically

```
realm
        inside          ns1.lucent.com, ns2.lucent.com, ns3.lucent.com
        outside         ns.isp1.net, ns.isp2.net, ns.isp3.net
switch
        inside  any     bell-labs.com
        inside  any     lucent.com
        inside  any     merger.com
        inside  any     spinoff.com
        inside  any     localhost
        inside  any     135.in-addr.arpa
        inside  any     11.192.in-addr.arpa
        outside any     *
filter  outside block   * NS *
        outside block   * A 127/8
        outside insist  * 28800 MX 100 mailgw1.lucent.com
        outside insist  * 28800 MX 150 mailgw2.lucent.com
        outside insist  * 28800 MX 200 mailgw3.lucent.com
```

Figure 2: Sample dnsproxy.conf.

depending on the question. This server was also written in-house. The design is to have a simple server which creates consistent A and PTR records depending on the question, and delegate the appropriate *in-addr.arpa* domains toward the bogus nameserver. The source code for this server is not available at this time, although its implementation is nearly trivial.

DNS Implementation

The DNS implementation entailed adding two new classes of servers to the Lucent infrastructure. First dnsproxy servers were setup at the backbone. These servers were configured with a configuration file similar to that in Figure 2. Two *realms*, which specify a collection of name servers that can be queried, are defined. One realm is for internal lookups and another for Internet lookups. The *switch* directive determines the realm to query for each domain, based on resource record type. We explicitly tell the dnsproxy to query internal name servers for both our own internal domains and also those that we have special connectivity arrangements with. We also absorb all the in-addr.arpa domains for Lucent's networks, as we (internally) are authoritative for them. All other queries are directed to external name servers.

Filter rules assign actions to perform on certain responses. Our configuration prevents the retrieval of NS records because internal hosts shouldn't be concerned with real Internet name servers. All queries should be directed up through the internal DNS hierarchy and eventually end up at the dnsproxy server. Additionally, we don't accept suspicious addresses from the outside. Finally, all internal queries for MX records are answered by the dnsproxy itself, in the form of an *insist* directive. The insist directive was a customization due to our Internet mail handling policy.

The other major change made to our DNS infrastructure was by modifying the behavior of the root name servers. Under the old architecture, the internal roots were authoritative for everything, period. Now, we are still self-rooted, but the root name servers have delegated most of the top level domains to the dnsproxy server.

Our internal root servers handle name serving for *lucent.com*, and some other second-level domain names. All other top-level domains are delegated to the dnsproxy, as well as the remainder of the *com* domain. This allows us to handle special cases by direct local name serving or internal delegation, while the dnsproxy handles everything else.

Hard Lessons

At this point, it may appear as if everything has been fairly well thought out and that implementation probably went without a hitch. Unfortunately, this is not the case. During the deployment, Lucent's corporate backbone was brought to it's knees no less than three times (during off hours and scheduled change windows of course, but still...)

The largest problem we had was by far the unpredictably of changes made at the Internet perimeter and the backbone on downstream routing domains. Specifically, propagation of default routing information was made quite difficult by the pockets of our intranet still using IGRP. The notion of a default route is a bit different than that of OSPF or BGP.

To work around IGRP's peculiarities, we had to modify the notion of the IGRP gateway of last resort. Originally, the core backbone was announced as the gateway of last resort. The problem with this was that the backbone routers running both IGRP and OSPF would use IGRP's default gateway, which was the directly connected backbone network. This was remedied by pointing the new default network to an upstream, Internet sourced route which caused traffic to be forwarded to the innie-outie routers instead of being dropped at the IGRP backbone router.

We also ran into some issues with the new DNS architecture. The original implementation of dnsproxy allowed for the rewriting of responses for MX queries to our own internal mail gateways. This worked fine except for those sites which didn't have MX records, but did have valid A records. The query would go out, but no MX query would be returned to rewrite, so internal mailers would lookup address record information. The internal mail host would then attempt to connect to the SMTP port of the destination host which would be blocked by the firewall. Thanks to the persistence of most mailers, the message would be deferred and spooled for later delivery. This caused mail spool directories to get quite large for awhile until a workaround was implemented. Additional modifications were required to dnsproxy in order to send back a forged MX response for any query. This caused all outbound mail to be routed to the corporate mail gateways (even to non-existent domains). Poorly addressed mail was simply bounced at the gateway instead of on the sending host.

Another issue that came up with the DNS was the failure of some internal name servers to resolve Internet names. This would happen during the handling of some recursive queries. Since the dnsproxy server filters requests for NS records, internal name servers would be unable to recursively follow delegations and answer the query. This problem was remedied by configuring our internal name servers with *forwarder* directives which pointed unknown queries back to our root servers (which could directly ask the dnsproxy server).

Where To Go From Here

At this point, we have implemented the primary/backup routing architecture and deployed the dnsproxy-based DNS infrastructure. The DNS architecture has served well and isn't likely to change any

time soon. On the other hand, the routing design has plenty of room for improvement.

Many of the constraints that led us to the existing routing design were historical, organizational, or otherwise political designs. There has been a realization within Lucent's WAN engineering groups that things need to change in order to manage the network effectively as well as supporting business needs. Listed below are a few of the directions that will improve the efficiency, reliability, and maintainability of our network.

1. Using Full BGP Routing

By receiving the full Internet routing table via BGP, we could route Internet-bound traffic out the best-path gateway. Recall, the reason this wasn't done initially because our existing topology didn't permit the internal BGP peers to be directly connected. Changes described below will soon change this.

BGP will readily determine the best exit-point to any given Internet host, but this is only half of the problem. Also recall that our stateful packet-filtering firewalls generate a requirement for symmetric routing. The difficult part isn't the best-path selection, rather assuring that the best-path to the Internet returns through the same firewall on the return path. Possible solutions to this problem are still in the whiteboard stage.

2. ATM Backbone Infrastructure

The existing backbone infrastructure has been converted to ATM, so we can now provide physical connectivity between our innie-outie routers. Currently the innie-outie routers are connected to the backbone via a fast ethernet switch. By replacing this with an ATM switch and interface, we can configure virtual circuits between the innie-outie routers at the various backbone locations.

3. BGP as the Backbone Routing Protocol

This is being investigated as a possibility to help manage the Internet routing as well as the plethora of mergers and acquisitions that occur regularly. The biggest advantage with respect to Internet routing is that internal routes could be tagged much easier upon redistribution into BGP than into OSPF. This tagging can be used to mark certain routes for specific egress points, which is necessary for symmetric Internet routing. This proposal is still being investigated.

Author Information

D. Brian Larkins is a Member of the Technical Staff at Lucent Technologies, where he is an engineer with the CIO Internet Services and Support Group. His responsibilities include architecture and design for securely connecting the Lucent WAN to the Internet, as well as engineering Lucent's e-commerce infrastructure. Brian began working on web technologies and Internet security in 1994 for AT&T's *www.att.com* site. He holds a B.S. in C.S. from the Ohio State University. Brian can be reached at <brian@lucent.com>.

References

[Bellovin95] Bellovin, S., "Using the Domain Name System for System Break-Ins," *Proceedings of the 5th Usenix Security Symposium*, 1995.

[Chandra96] Chandra, R., "Bates, T., BGP Route Reflection: an Alternative to Full Mesh IBGP," *RFC 1966*, June, 1996.

[Chandra97] Chandra, R., *Introduction to Border Gateway Protocol*, presentation at Cisco Networkers, June, 1997.

[Cheswick96] Cheswick, W., "A DNS Filter and Switch for Packet-filtering Gateways," *Proceedings of the 6th Usenix Security Symposium*, 1996.

[Cheswick94] Cheswick, W., *Firewalls and Internet Security; Repelling the Wily Hacker*, Addison-Wesley, 1994.

[Fuller93] Fuller, V., Li, T., Yu, J., Varadhan, K., "Classless Interdomain Routing: an Address Assignment and Aggregation Strategy," *RFC 1519*, September, 1993.

[Halabi97] Halabi, Bassam, *Internet Routing Architectures*, Cisco Press, Indianapolis, IN, 1997.

[Halabi95] Halabi, Bassam, *BGP4 Case Studies/Tutorial*, Cisco Systems, 1995.

[Huitema95] Huitema, Christian, *Routing in the Internet*, Prentice Hall, Englewood Cliffs, N.J., 1995.

[Khan97] Khan, A., *Designing and Troubleshooting OSPF Networks*, presentation at Cisco Networkers, June 1997.

[Limoncelli99] Limoncelli, T., "Tricks you can do if your Firewall is a Bridge," *Proceedings of the 1st Usenix Conference on Network Administration*, 1999.

[Moy94] Moy, John., "OSPF Version 2," *RFC 1583*, March, 1994.

[Raza97] Raza, K., *Configuring BGP Networks*, presentation at Cisco Networkers, June, 1997.

[Rekhter95a] Rekhter, Y., Gross, P. (editors), "Application of the Border Gateway Protocol in the Internet," *RFC 1772*, March, 1995.

[Rekhter95b] Rekhter, I., Li, T., "A Border Gateway Protocol (BGP-4)," *RFC 1771*, March, 1995.

[Rekhter94] Rekhter, Y., Gross, P., "BGP4/IDRP for IP – OSPF Interaction," *RFC 1745*, December, 1994

[Stevens94] Stevens, W. Richard., *TCP/IP Illustrated, Vol. 1*, Addison Wesley, Reading, MA, 1994.

[Umali97] Umali, T. (editor), *Lucent Technologies IP Data Network Architecture Document*, Internal Memorandum, 1997.

[Umali96] Umali, T. (editor), *Lucent Technologies IP Routing Implementation Document,* Internal Memorandum, 1996.

[Varadhan92] Varadhan, K., "BGP OSPF Interaction," *RFC 1364*, September 1992.

Moat: a Virtual Private Network Appliance and Services Platform

John S. Denker, Steven M. Bellovin – AT&T Laboratories
Hugh Daniel – FreeS/WAN Project
Nancy L. Mintz, Tom Killian, and Mark A. Plotnick – AT&T Laboratories

ABSTRACT

We have implemented a system for virtual private networking, with special attention to the needs of telecommuters. In particular, we used off-the-shelf hardware and open-source software to create a platform to provide IP security and other services for in-home networks.

Our experience has taught us a number of things about the scalability of the FreeS/WAN IPsec system, about the widespread mis-handling of path-MTU discovery on the internet, and about the implications of tunnels on the basic architecture of the network.

Additional Keywords: VPN, Linux, Residential Gateway, MSS, fragmentation.

Overview

Extending the Network

Suppose we have some customers who are affiliated with a corporation[1] that has a good local-area network, possibly even a wide-area network. Further suppose that there are secondary locations that have not heretofore been served by the corporate network; these could include the customers' homes, or a smallish branch office, or whatever.

In many cases, linking the secondary location to the corporate network is highly desirable. The customers may use the secondary location in order to save the time, money, and risk associated with commuting to

Physically versus Virtually Private Networks

Over the years, we have used several different methods for connecting the secondary location to the main network.

One approach was to use a *non-private network*. For example, the customers could use a local ISP to establish a link from their PCs to the Internet. From there, they could telnet to some corporate portal. This has several drawbacks. For one thing, this simple configuration exposes their PCs to attack from all the hackers on the Internet. To fend off such attacks, they would need some sort of firewall. Another drawback is that there are multiple points where their data (including sensitive information such as passwords) could be read by eavesdroppers, and even possibly altered in transit.

Another approach was to arrange it so that each corporation's data moved over physically separate wires. This is sometimes called a *Physically Private Network*. We found such systems to have many drawbacks. Either the secondary locations needed to make long-distance calls, or multiple strategically-placed modem banks were required. Each modem bank required an expensive dedicated "backhaul" link to the main location. Furthermore, it is getting harder and harder to physically protect such links against tampering.

Nowadays, the best approach in most cases is to transform a non-private network into a virtually private network (VPN) using software. The rest of this paper is devoted to explaining how this is done.

Physical View

A typical telecommuting situation is shown in Figure 1. (Many variations and extensions are possible, some of which will be discussed below.)

The networks drawn with double lines constitute the private network. The objective is to give all machines on the private network a reasonable degree of protection against attacks coming from anywhere outside.

In typical usage, a packet goes from one of the clients on the in-home network, through the moat, via the internet, through the security portal, to a host on the corporate network. The moat and the security portal are the primary subjects of this paper.

In Figure 1, if we dared to connect the client machines directly to the internet (instead of going through the moat), we would be exposed to all sorts of attacks, including the following:

- For starters, a typical machine in our address space is routinely attacked every couple of days by probes looking for various well-known security loopholes. Our cable provider filters out attacks against the ms-networking file-sharing ports; otherwise the number of attacks would be even larger.

[1] We will extend the term "corporation" to include universities, government agencies, and other such entities, even if they don't meet the precise legal definition of corporation.

- Secondly, there is the possibility that evildoers could compromise one of the routers or other machines along the way.
- A relatively remote possibility is that a neighbor could attack the signal on the neighborhood cable. This is harder than you might think, because of security features in the cable modems. Each modem (unlike, say, an ethernet tap) knows what IP addresses it is supposed to serve, and is programmed to not pass traffic intended for other modems. This is enforced with weak link-level encryption.

There are two ways to look at how our system protects the private networks. We begin by discussing the physical viewpoint, as diagrammed in Figure 1.

On the home network, the client machines (the PC, the Mac, etc.) are configured so that the moat is their default router. When an IP packet (which we will call a raw IP packet) from the client machine arrives at the moat, it is encrypted. This encrypted data is then encapsulated ("put into an envelope") and sent via the internet to the security portal at corporate headquarters. The security portal removes the encrypted data from the envelope, decrypts it, and sends the raw IP data on its way via the corporate network.

Data flowing in the other direction is treated the same way.

Because of this system, a hacker cannot determine the raw contents of the packets flowing between the home LAN and the corporate network. Furthermore, the hacker cannot determine the identities of the client machines or the corporate hosts, or even how many of them there are. The only traffic that flows over the public internet consists of packets from the moat to the security portal and vice versa. All such packets use IP protocol 50 (ESP, i.e., Encapsulated

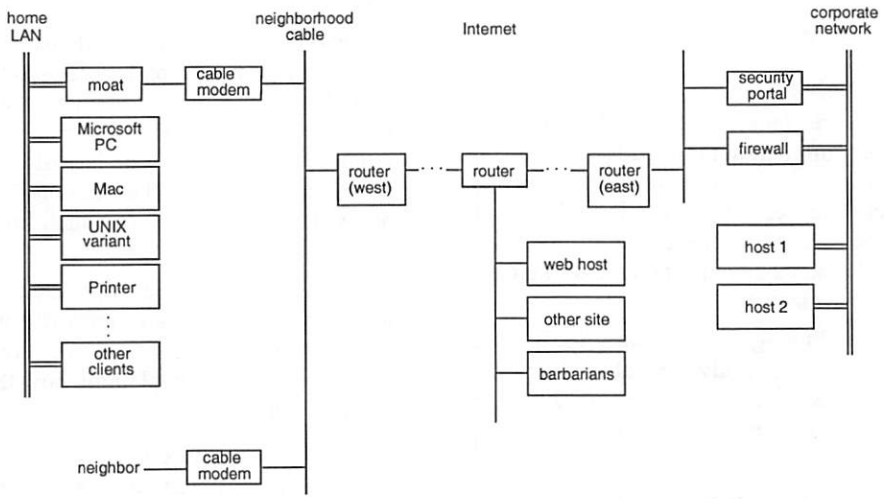


Figure 1: Telecommuting Setup – Physical View.

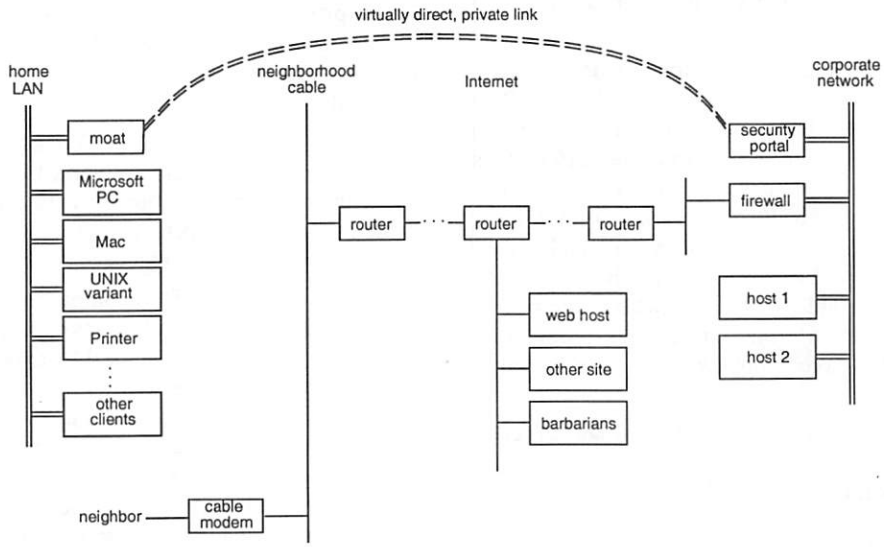


Figure 2: Telecommuting Setup – Virtual View.

Security Payload) so an attacker cannot even determine the IP protocols that the raw packets are using.

A typical machine on the public internet will have no route to the client PC except through the corporate firewall. Even if an attacker guesses that the moat is in a position to forward traffic to the client PC, the moat refuses to forward traffic that doesn't come from the corporate security portal.

Virtual View

Another way to look at this is the virtual viewpoint, as diagrammed in Figure 2. It appears to the users that there is a virtually direct, private connection from the moat to the security portal.

Indeed, the traceroute command conforms to the virtual view, reporting that there is a single hop between the moat the the security portal, no matter how many physical hops there are in Figure 1.

In the setup shown in Figure 2, the only way a client machine can contact a server on the public internet is to go out through the firewall at corporate headquarters.[2] In the other direction, a machine on the public internet cannot initiate a telnet session to any machine on the private network, because that is disallowed by the corporate firewall.

Goals

This project began as a research project, but quickly scaled up when a group within AT&T needed a large number of VPN systems. They found out three weeks before the deployment deadline that the commercially available hardware and software systems that they had expected to use were unreliable and/or unavailable in sufficient quantity. Our main goals, in approximate order of importance, were:

- available on time.
- reliable.
- suitable for a heterogeneous home network (not just a single machine running windows 98).
- easy to install, administer, and manage.
- scalable (as discussed in section "Scalability Issues").
- capable of matching the speed of the cable modem.
- usable as platform for further research in networking and services, e.g., green light for alerting for email).
- affordable.
- quiet (otherwise it will get turned off).
- physically small.

We met all these goals except that the moats in the first batch were not quite as small as we wished.

Ingredients

Hardware

For the first-generation moats, we chose the following configuration:

- Commercial off-the-shelf PC platform.
- No keyboard, no screen, and no video subsystem.
- No sound subsystem and no CDROM drive.
- Two ethernet network interface cards.
- Minimal RAM (16 MB, which is about 2x more than needed).
- Cheapest possible hard disk (4 GB, about 100x more than needed).
- Cheapest possible CPU (K6, 300 MHz).
- Power supply with ultra-quiet fan.
- Floppy drive.
- Serial port.
- BIOS that is happy to boot with no keyboard or video card.

We arranged with the PC manufacturer to have the whole batch built with the desired software (listed below) and our public keys pre-installed.

In the second-generation moats, we used an LS-120 superfloppy in place of the hard disk and floppy. Since the disk is used exceedingly rarely (essentially for boot-up only), wearing out the disk is not an issue.

Software

Running on the typical moat we have

- Linux – operating system.
- FreeS/WAN – IP security system.
- ssh – secure shell.
- configuration scripts.
- network monitoring scripts.
- dhcp client – to get the moat's wild-side IP address from the ISP.
- dhcpd* – to assign IP addresses to client machines.
- xntpd* – time-of-day setting system.
- named* – Domain Name System secondary.

where the three services marked with (*) are accessible from the private-side interface (facing the in-home network) and not from the wild-side interface (facing the cable modem).

Xntpd is more important than you might think; it allows the logfiles on the moat and the security portal to be compared.

The reason why the moat provides DHCP and DNS service on its own, rather than simply forwarding such requests through the IPsec tunnel, is that we want users to be able to use their in-home networks even if the wide-area network is temporarily down.

Software for the moat is cross-compiled; that is, the moat does not contain its own development environment. On the machine where we do the compilations, we rely on gcc and the gnu development tools, and cvs.

It is also worth noting that the services listed above are the only ones running. In particular the moat provides:

[2] There are other ways to do this, as will be discussed below, but this method is particularly easy to implement, and is easily shown to be compliant with corporate firewall policy.

- no web server,
- no telnetd,
- no ftpd, and
- none of the basic inetd services. The moat literally will not give you the time of day (port 37).

Scalability Issues

A stated goal of the FreeS/WAN project is to encourage very wide adoption of IP security. Similarly, AT&T is committed to providing secure IP service over shared media (e.g., cable modems) on a large scale. To us, if it's not scalable, it's not interesting.

Therefore it is not sufficient to have an IPsec implementation that supports just a few tunnels per portal, lovingly configured by an expert.

The system must be database driven, so that hundreds or thousands of customers can be transferred from one system to another, without requiring laborious manual re-entry of data.

The system must not assume that the configuration is known and unchanging. When you have hundreds or thousands of tunnels per portal, something is always changing.

At our behest, the FreeS/WAN team added many "scalability" features to the IPsec system. These include:

- The system can add, delete, or reconfigure tunnels on the fly. The configuration file, which in small-scale systems could be a single flat file, implements an include directive which allows us to implement a database with one record per moat. Note that we are using the Linux directory/file system as our database, using one file per record.
- If one of the tunnels cannot be brought up, the system does not hang waiting for it, but goes on to the next.

Configuration Procedure

Databases

We have a simple database that lists all the moat customers. The key fields include

- the wild-side IP address (in case the ISP statically assigns one to this customer) or an indication that the wild-side address will be assigned dynamically via DHCP.
- the range of private-side IP addresses (which we assign arbitrarily).
- the email address to which we send the "welcome" message and instructions for configuring the client machines.

We have a collection of shell scripts (about 1000 lines total) that extracts the required information from the database and configures the moat. Without these scripts the configuration would be quite laborious and error prone. There are more than a dozen configuration files that are affected.

Identification Without Static IP Addresses

The current versions of the FreeS/WAN IPsec package provide only one way for a moat to identify itself, namely, its wild-side IP address. This is a disaster in cases where the ISP assigns a non-constant address to the moat. To overcome this limitation, we devised a complex procedure:

- The moat gets its wild-side IP address *du jour* from the ISP via DHCP.
- The moat rewrites its IPsec configuration files accordingly.
- The moat contacts the security portal using ssh. Note that this does not require the IPsec tunnel to be operational.
- The security portal infers the moat identification based on what ssh key the moat presents, and can see what IP address the moat is using.
- The security portal rewrites its IPsec configuration files accordingly.
- Both sides bring up the tunnel using the new configuration.

The IPsec RFCs envision other forms of identification, such as using the Fully Qualified Domain Name (FQDN) but they have not yet been implemented in FreeS/WAN.

Comparison Against Alternatives

- We have seen hardware solutions that provide features comparable to the moat, but they were more than twice the price of the moat, and were hard to administer.
- Our cable provider charges extra for more than one IP address, and won't provide more than three IP addresses at any price. This affects all VPNs implemented in software.
- We have evaluated software solutions. The leading contender...
  - is not reliable for windows 95. Even on windows 98 it has been known to crash in such a way as to wipe out your C drive.
  - is not available for non-microsoft systems.
  - requires laborious and error-prone installation per PC.
  - requires laborious re-configuration every time the ISP changes the client's wild-side IP address. Note that some ISPs change the IP address multiple times per week.
- The microsoft implementation of PPTP has half a dozen known security holes.
- We probably could have achieved comparable results using other operating systems and/or IPsec packages. There are so many of them that we could not possibly evaluate them all.

Multiple In-Home Networks

In many homes, there are multiple computers, and having them all connected to the same virtual

private network may not be the optimal configuration.

- For example, there may be kids in the house who want to surf the internet. The kids have their own computer(s) and should not be using the corporate work-at-home computers.
- As another example, there may be two or more adults with different employers, each of which needs a separate VPN.

Brute-Force Solutions

Right now the kid-net feature can be implemented as shown in Figure 3, where PC-a is connected directly to the wild internet. The user puts a hub between the moat and the cable modem, and arranges with the cable provider to get $N + 1$ IP addresses (one for the moat, plus one for each of the non-corporate client machines). Typically the cable provider imposes a modest charge for each additional IP address, and some providers impose a limit of three IP addresses per customer, or some other low limit.

A lame approximation of this feature can be achieved without paying for a second IP address. You take turns plugging the moat or the kids' computer directly to the cable modem. The cable modem must be reset each time (because it otherwise remembers the Media-Access address of the host to which it is connected, and refuses to talk to any other host). Each reset takes about three minutes. Sigh.

Moat-Based Solutions

A more-elegant solution to the same problem is shown in Figure 4. The wild-side interface of the moat is physically capable of accessing the entire internet. We can put another connector on the moat, implementing another subnet that the kids could use. An advantage of this over the previous solution is that the moat would offer this subnet some minimal firewall service.

A similar solution, again employing multiple interfaces on the moat, could provide for multiple VPNs.

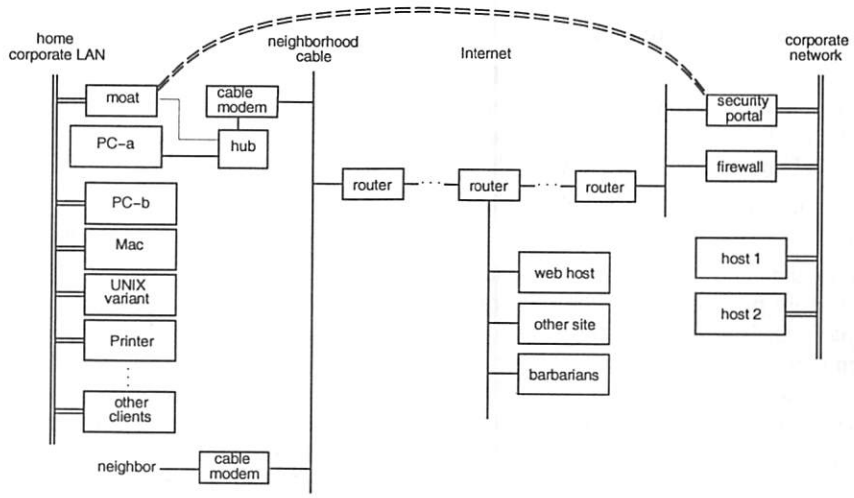


Figure 3: Multi-Net Hardware Solution.

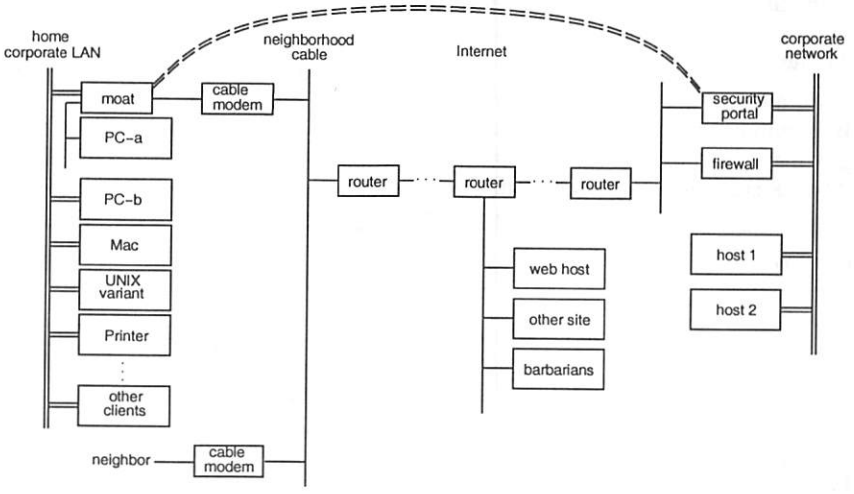


Figure 4: Multi-Net Software Solution.

This is 100% possible in principle. Like many things, doing it badly would be easy, but doing it well is not so easy. With N interfaces on the moat, there are N factorial paths that packets could take, most of which are disallowed. For $N > 2$, a solution that preserves security with high assurance would require more careful work than we have yet been able to give, so this is still in the experimental stage.

Distributed Firewalls

We now consider the internet connectivity of PC-b as shown in Figure 4. This machine (unlike PC-a) has access to the corporate VPN.

Given a sufficiently fast link to the main corporate network, it works just fine for traffic between PC-b and generic hosts on the public internet to go through the corporate firewall. However, it would be more elegant, and in some cases more efficient, if the moat itself could implement corporate firewall policy itself, and route such traffic to/from the internet directly. This is discussed in [3].

Tunneling & Fragmentation

It is quite possible for a tunnel that is fully RFC-compliant to be unable to interoperate with a very large percentage of the sites on the internet, because of fragmentation and MTU problems, as we now explain.

Keep in mind that the objective is reliable and efficient communication. That's all.

As a means to that end, it is better to send a smaller number of large packets, rather than a larger number of small packets. If a packet is fragmented in transit and reassembled before delivery, it magnifies the effect of packet loss in transit. That is because the higher levels of the protocol, which are responsible for retransmission, are forced to retransmit the whole packet, not just the fragment that got lost.

Therefore it is *sometimes* true that the most-efficient packet size is the largest size that can be transmitted without fragmentation, i.e., the *path MTU* (Maximum Transmission Unit). But this is not always true, as discussed below.

As a means to attempt path-MTU discovery, hosts often begin by sending large packets with the DF bit set, and seeing if they get through. But remember this trick is at least two assumptions removed from the actual goal of reliable and efficient communication. See http://www.ietf.org/rfc/rfc1191.html for details.

Many TCP clients (notably microsoft) make an optimistic guess and set their initial MSS (Maximum Segment Size) to the largest plausible value. This is perfectly proper, and should typically result in efficiency if other players do their part. This initial guess is necessarily made with no knowledge of the actual path-MTU.

The large initial MSS makes it likely that early in the session, packets will be sent that exceed the MTU of some router along the path – especially when there is encapsulation going on at some point, such as an IPsec tunnel. Remember that the MSS concept is applicable at the TCP layer, while the MTU concept is applicable at a much lower layer. The assumption that an MSS of size X will correspond to an MTU of size $X + 40$ is invalidated by the overhead bytes introduced by the encapsulation.

Suppose an oversized packet (with the DF bit set) arrives at a router. The RFC says "In this case the gateway must discard the datagram and may return a destination unreachable message." Specifically this message is ICMP type 3 code 4 and it explains that fragmentation is needed and suggests a new packet size. See http://www.ietf.org/rfc/rfc792.html for details.

Note that the frag-needed messages are *optional* according to the RFC.

We note that path-MTU discovery without them is at best rather inefficient. But in real life, the situation is much worse than that. We observe that the vast majority of the world's web servers improperly assume that the routers *must* return a frag-needed message. The microsoft web site is one of the few that is both efficient and robust... efficient in that it starts out by sending large packets, and robust that it will (even in the absence of frag-needed messages) reduce its MSS if large packets don't get through. See observations section below.

There are some firewalls (the Firewall-One brand in particular, and quite likely others) that in their usual configuration do not pass the ICMP frag-needed datagrams. We consider this a weakness in the firewalls. This is a pain in the neck to fix, but overcomes

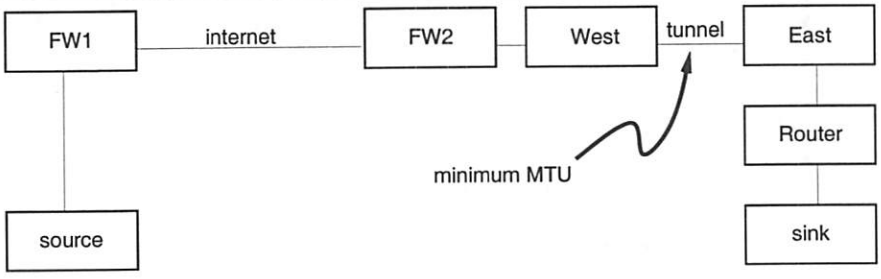


Figure 5: Interior Minimum MTU.

the naughty behavior of about half the world's web sites; see details below.

What's much much worse is that the persons administering typical IPsec tunnels cannot in general fix the frag-needed problem by fixing their own firewalls. Consider the situation shown in Figure 5, where there is a tunnel between West and East. Suppose you control FW2 and everything to the right thereof, whereas FW1 and everything to the left thereof belong to somebody else like aol.com.

It appears that something like 15% of the server-sites in the world assume that there will be "no interior minima" in the MTUs along a path. They create their own black hole, and then improperly fail to perform black-hole detection. Your clients will hang when trying to contact such servers, and there 's almost nothing you can do about it.

Similarly it appears that more than half of the world's sites improperly require the frag-needed messages when there are interior minima.

Note that a verrrry large fraction of the world's clients blissfully (and properly!) negotiate for the biggest MSS they can get.

Finally note that we want the tunnel to interoperate with the world as it actually exists. This is a stricter requirement than merely complying with the RFCs.

Therefore our policy is to have the tunnel implement a virtually-large MTU. This will in some cases require that packets (with DF set) that arrive at one portal (West) be encapsulated in multiple envelopes. When these envelopes arrive at the other portal (East) they will be re-assembled so that an unfragmented packet can be sent on its way toward the final destination (sink).

This policy comes as a shock to some people, because it eliminates any possibility of the source/sink pair being able to discover the effective MTU of the path inside the tunnel. However this is the way it has to be. Suppose for instance that the path inside the tunnel had (at the real-IP level) an MTU equal to the smallest permissible internet packet. If we tried to force the incoming raw packet to be sent in a single envelope, there would be no room for the encapsulation header overhead, and therefore no connectivity at all. An extreme way to make the same point is this: suppose we are sending IP over ATM, which uses 48-byte packets. Setting the DF bit cannot possibly prevent fragmentation of the IP packet into multiple ATM cells.

The basic, essential function of the DF bit is to ensure that packets that leave the source with the DF bit arrive at the destination in one piece. This is important, because not all hosts are capable of reassembling fragments. Note that any packets that are fragmented on entrance to the tunnel are reassembled upon exit, thereby upholding this essential meaning of the DF bit.

Note that this virtually-large MTU is consistent with the fact that (at the abstract level) there is only one hop between the two ends of the tunnel (as reported by, e.g., traceroute) no matter how many hops there are (at the real-IP level) inside the tunnel.

This inability of the endpoints to discover the MTU of the tunnel will in some cases lead to reduced efficiency. In our application (and we believe in most applications) this inefficiency is vastly preferable to

Site	MTU Observation
www.microsoft.com	efficient and robust; large initial MSS but notches down
www.ibm.com	inefficient: always requests a small MSS
www.snap.com	inefficient: always requests a small MSS
www.toad.com	inefficient: never sets DF, always sends small packets
www.sandelman.ottawa.on.ca	inefficient: never sets DF, always sends small packets
www.hotbot.com	semi-naughty: eventually reduces MSS, but is horribly slow about it
www.netscape.com	naughty
www.altavista.com	naughty
www.yahoo.com	naughty
www.clinet.fi	naughty
www.sgi.com	naughty
www.psi.net	naughty
www.cygnus.com	naughty
www.quintillion.com	naughty
akpublic.research.att.com	naughty
www.aol.com	incorrigible
www.intel.com	incorrigible
www.compaq.com	incorrigible

Table 1: MTU bug observations.

the alternative, which is complete inability to reach incorrigible sites such as aol.com and intel.com.

Observations

Terminology: "naughty" means the server site (meaning the source/FW1 combination) does not recover properly when we start out with an MSS that cannot be supported by the actual path-MTU, unless an ICMP frag-needed message is emitted by FW2. Similarly "incorrigible" means that it does not recover even if the frag-needed messages are being generated and passed by FW2; presumably they are being blocked by FW1 or some such.

Table 1 gives a partial list of servers we've checked.

Summary: Tunnel MTU

- There are lots of firewalls out there that eat ICMP.
- There are lots of naughty sites out there that act as if gateways *must* generate frag-needed messages, even though the RFC doesn't require it.
- There are more than a few incorrigible sites out there which (perhaps because of their own firewalls) don't do the right thing even when frag-needed messages are being sent toward them.
- There are clients (notably microsoft) that negotiate for the largest possible MSS. The source and sink make an initial guess based on information about the MTU of the first link at each end of the path, but they have no initial knowledge of the real path-MTU. This behavior appears to be fully compliant with the RFCs.
- The foregoing can be summed up as follows: many sites make the implicit, improper assumption that the path will have no interior minima.
- Tunneling software is quite likely to break that assumption.
- In special cases, it may be possible to get enough control over the clients so that the tunnel can advertise its real MTU, thereby gaining efficiency... but this cannot be the default.
- More generally, to achieve a good level of interoperation with the world as it exists, warts and all, a tunnel needs to have a virtually-large MTU.

Routing

The main points of this section are:

- Every portal is a router, and needs to act like one.
- The existence of tunnels requires us to re-think basic internet architecture.

Before explaining these points, we must distinguish a couple of concepts: The kernel (Linux in this case) implements a "kernel routing table". This allows packets within a given host to be routed to the correct interface (eth0, eth1, et cetera). Given a properly set up routing table, the host can function as a gateway, so packets that come in on one interface can be forwarded to another. We will call this level of functionality a *micro* router.

A fully-featured internet router requires other functions. It needs to run a program such as routed(8). That is, it needs to send and receive routing messages, so that it and its peers can keep their routing tables up to date in the face of ever-changing network connectivity. A host with this level of functionality we will call a *macro* router.

Layers of Routing

Next, we must distinguish three or four levels of routing that must occur:

First, there is routing that occurs well below the IPsec layer. Consider the routers shown in Figure 1. They must maintain their routing tables to reflect the routes that connect them. This occurs at the raw-IP layer, and the portals at the ends of the IPsec tunnels generally do not and cannot know anything about this.

Next, there is routing that should occur below the IPsec layer but which is actually done at the IPsec level. The current FreeS/WAN implementation does not trust the kernel routing tables to send the IPsec traffic over the right route. Therefore, as part of the IPsec configuration, FreeS/WAN must be told about the "next-hop" router. That is, in Figure 1, the moat must be told the IP address of the "west" router, and the corporate security portal must be told the IP address of the "east" router.

Next, there is routing that must occur at the IPsec layer. The linux 2.0xx kernel routing mechanism can only route traffic based on the destination IP address. However, the IPsec specifications require that decisions about how to encrypt, and how to route traffic, must be made on an assortment of "selectors" including destination address, source address, protocol type, et cetera. The current version of FreeS/WAN only implements the source and destination selectors.

Another type of routing occurs at or above the IPsec layer. This occurs when a given host has more than one tunnel that serves the same subnet. In Figure 1, imagine that there were redundant security portals on the corporate wide-area network, located at a number of different sites. The IPsec implementation on the moat would need to choose which of these tunnels to use. This decision would be based on information from lower protocol layers, i.e., how efficiently each tunnel can carry traffic, whether it is up at the moment, et cetera.

Finally, there is routing that must occur above the IPsec layer. A portal creates a route to the machines on the far end of the tunnel. In general, the portal should act like a macro-router; i.e., it should advertise this route. For instance, in the case of multiple portals just described, hosts and routers on the corporate network would have a choice of which portal to use when sending traffic to the moat. This would depend not only on the efficiency of the tunnel (once

the traffic reached the portal) but on how efficiently a given host could reach a given portal.

The current version of FreeS/WAN cannot handle redundant tunnels. It does not perform high-level macro-routing (i.e., advertising its routes). It does not gracefully handle low-level macro-routing (e.g., changes in the address of the "west" router). These are hard problems, and are very high on our wish-list.

Grand Implications

Consider the following chain of implications:

- There are many good reasons to widely deploy IPsec.
- IPsec requires tunneling. This is the first good reason for really wide application of tunneling.
- The existence of tunnels requires a re-think of the basic design of the internet.

The existing procedures for macro-routing do not appear to scale very well. There are too many routes, and too many updates to the routing tables.

The deployment of large numbers of tunnels could greatly increase the number of routes. This can be expected to put additional stress on the routers.

Furthermore, we must examine the notion that the tunnel is layered on top of the raw-IP link. Link failures break this notion, cutting across the layer boundaries. If (in Figure 1) the route from "west" to "east" changes, that's no big deal, but if "west" totally loses its connection to "east", then the moats (possibly a very great number of moats) must be notified.

The technique of sending routine enquiries ("keep-alive" messages) to check on the health of a path scales exceedingly poorly. If every endpoint of every conversation tried to do this, very soon the entire bandwidth of the net would be used for keep-alive messages. A better strategy is for each node to keep track of the health of its immediate neighbors. Then, if there is an outage, it can send appropriate notifications using higher-level protocols.

The present situation has at some interesting upsides. For example, the IPsec header contains a sequence number. If packets arrive out of sequence, it is a sensitive indicator of loss – more sensitive than the sequencers built into the higher-level protocols. Having a good loss indicator is very important, since loss is practically the only usable indicator of congestion.

It may be possible to exploit this and other properties of the IPsec protocol to solve some problems. Since the IPsec protocol is not 100% set in stone at this point, we have what may be a once-in-lifetime opportunity to design in additional features to solve important problems that heretofore seemed only tangentially (or less) related to the basic objectives of IPsec.

Other Lessons Learned

- Until recently, the ethernet drivers for Linux were quite intolerant of small variations in the ethernet cards. We found many cases where a driver worked OK with a name-brand card, but failed miserably with clones, even clones that work perfectly well under MS windows.

  Furthermore, we found that even with name-brand cards, many of the drivers failed under high-load conditions. This was particularly serious because the FreeS/WAN Linux-IPsec system appears to stress cards and drivers in ways that ordinary network traffic does not.

  Recent drivers seem much better behaved.
- Linux is remarkably vulnerable to having its power turned off when the filesystems have not been cleanly unmounted.

  Since the moat has no screen or keyboard, a filesystem inconsistency that cannot be fixed by "fsck -a" or other automatic procedure incapacitates the system totally and permanently. We fixed this by mounting a ramdisk on /, making /usr read-only, and arranging that if /var fails fsck, it is restored from a read-only backup.
- In the home environment, people are quite intolerant of machines with noisy fans.

Performance

The round-trip "ping" time between a typical client machine and a typical host on the corporate network is routinely less than 25 milliseconds for short packets and less than 85 milliseconds for 1400 byte packets. Almost all of this time is attributable to the link between the moat and the security portal at the real-IP level; the encryption and encapsulation takes at most 1 ms for the short packets and at most 15 ms for the long packets.

On a laboratory link with negligible delay at the real-IP level, the system is observed to sustain a throughput of 6.9 megabits per second. This is at or near the limit imposed by the CPU cycles required to perform the triple-DES encryption of the data stream.

Hardware reliability has been good. During roughly 2000 moat-weeks of operation, we have not had any failures in the field except for one case where it appears the cable modem and moat were destroyed by a lightning strike, one hard-disk crash, and one transient outage attributed to overheating in an un-air-conditioned location in midsummer.

Software reliability has been good. We discovered one race condition that would cause a tunnel to fail with a probability of about 0.1% per day per moat. The FreeS/WAN authors quickly fixed this. The only large-scale outage resulted from a failure of syslogd of all things, due in turn to the Linux kernel's highly sub-optimal implementation of fsync(). It uses an N^2

algorithm, which cannot keep up when the log file gets too big.

Far and away the biggest performance problems are at the raw-IP layer. Remember, IPsec is layered on top of raw IP. If the lower layer is broken, there's not much the higher layers can do about it. We have learned that it is important to give the users (and support staff) enough access to the various layers that, when there is a problem, they can tell whether it is the raw-IP layer or the IPsec layer that needs attention.

Conclusions

- Our customers have been delighted with their moats. The combination of moat and cable modem gives the in-home LAN such good performance that customers usually find it indistinguishable from being at work.
- Our system administrators have been very happy with the moats, because they are more reliable and easier to administer than the alternatives.
- Using off-the-shelf hardware allowed us to meet a very tight time schedule at a very reasonable price.
- Using open software was a real joy. When things go wrong, you can fix them. When you want to add features, you can just do it.
- Our experience with the moats has given us a much greater understanding of which features are needed in an in-home service platform.

Acknowledgements

We are profoundly grateful to the authors of FreeS/WAN and the software mentioned in the "Software" subsection. Other folks who have made important contributions to the moat project include Norm Schryer, Sharon Gray, Sam Alexander, and Steven Gao.

References

[1] Technical information on IPsec can be found at http://www.ietf.org/rfc/rfc2401.txt and references therein.

[2] The FreeS/WAN home page is at http://www.xs4all.nl/~freeswan and includes links to detailed documentation.

[3] *Distributed Firewalls* by Steven M. Bellovin, http://www.research.att.com/~smb/papers/distfw.ps.

Automated Installation of Linux Systems Using YaST

Dirk Hohndel and Fabian Herschel – SuSE Rhein/Main AG

ABSTRACT

The paper describes how to allow a customized automated installation of Linux. This is possible via CDRom, network or tape, using a special boot disk that describes the system that should be set up and either standard SuSE Linux CDs, customized install CDs, an appropriately configured installation server, or a tape backup of an existing machine.

A control file on the boot disk and additional (optional) control files on the install medium specify which settings should be used and which packages should be installed. This includes settings like language, key table, network setup, hard disk partitioning, packages to install, etc.

After giving a quick overview of the syntax and capabilities of this installation method, considerations about how to plan the automated installation at larger sites are presented. This includes questions like the minimum default installation, criteria how to define classes of machines with identical additional setup and security concerns.

Finally, an extension of this methodology that allows system backups and automated restore of such system backups (somewhat similar to the mksysb-feature present in IBM's AIX) is presented.

Issues Of An Automated Installation Process

One of the major challenges of system administration at large sites is the installation of multiple machines that are similar, but not exact clones of each other. In some cases there are differences in the hardware, for example different network adapters or different disk sizes. At other sites the software requirements are very system specific, depending on the exact usage of the system, for example whether the system is a web server, needs a database engine or a user front end like X and KDE. As soon as more than a handful of system is involved, an automated installation procedure is needed, if not a must.

This procedure should be flexible enough to adapt to the slight differences between different machines and powerful enough to allow for choices to be made automatically during the install based on the system name, IP address, the hardware layout or other factors. And it should still be easy enough to handle so that neither a steep learning curve nor special in depth knowledge of the tools in use is required. It should give the system administrator a tool that doesn't add complexity but simplicity to the task of installing multiple machines.

Most of the large traditional Unix vendors have addressed this problem in some way, whether it is Sun's JumpStart (as described for example in [2]) or HP's Ignite/UX (see for example [3]) or other procedures like IBM's NIM [4]. Some work has been done on generic multi OS installation servers (for example in [5]), but little published material on that can be found. Another version of the Linux operating system (Red Hat) offers a different and in some aspects (e.g., multi disk systems, package configuration) less flexible way for automated installation. See [6] for details.

All these methodologies and products share some common parameters. First, the need for an install server or, in some cases, a special install medium. Second, some configuration information, normally provided as a configuration file, that is either provided locally on a floppy disk, or available via the net, indexed for example through the MAC address of the host that is being installed. Third some initial start of the boot process, so that the automated installation is initiated.

One advantage that the vendors just mentioned above have on their main hardware platforms is a tightly controlled set of hardware, usually from a very small number of vendors (if not themselves) only. And usually that hardware is built to rather tight standards that are designed for a networked environment. This is very different from the PC architecture that is addressed by Linux (and Solaris x86, for that matter). This architecture is extremely inhomogeneous and the standards are loosely interpreted, in general. The primary focus of most vendors is the Microsoft Windows world, not the requirements of a networked Unix environment. The minimal BIOS that almost all PCs come with has no default way to boot over the network. The choice of boot image and partition is very restricted, usually an additional boot loader is needed. There are several bus architectures and a mass of hard drive specifications and hard drive interface cards to support. Similarly, a huge number of (mutually incompatible, from a driver perspective) network boards needs to be taken into account already during the boot process.

What all this means is that the initial boot prior to the installation of such a PC system can already be a major challenge, if not a stumbling block. To solve this initial problem, the automated installation process described here is based on standard boot disks for SuSE Linux. This ensures wide testing of the boot process and compatibility with most PC hardware. Additionally it provides an installation environment that stays functional over time, even with newer versions of Linux, as it only uses documented interfaces and no undocumented customizations. The system administrator therefore needs to familiarize himself only once with this procedure and can continue to use it. This promise of continued usability is very important especially at larger sites, where the investment into an automated installation environment needs to amortize over time.

Changes To Be Done On The Boot Disk

YaST, the SuSE installation and configuration program [1], is usually interactive, guiding the user step by step through the installation (and later the configuration) of a SuSE Linux system.

In order to allow a completely "hands-off" installation, some additional setup is required. More precisely, a customized kernel that includes the drivers for the hardware that may be necessary during the install (which usually means the network card and the adapters that connect to the hard disks and in some cases the CD) has to be installed on a normal SuSE Linux boot disk in order to prepare the automated installation. Alternatively, you can also use the standard kernel and tell ldlinux.sys which modules (if necessary, with parameters) have to be loaded after booting the standard kernel.

Using special configuration files all necessary information for the installation is provided either on the boot disk or centrally on the install server. This includes information like the hostname, IP address, default gateway, nameserver, install server for the system to be installed. All this can of course be obtained using DHCP as well, but the ability to specify it using configuration files gives an additional degree of freedom, if, for example, no DHCP server is available in the current subnet. Additionally, the partition layout can be specified in various flexible ways. Either directly for each disk installed in the system (which requires some knowledge about how many disks the system has, what type they are, etc.) or generically with different layout classes depending on the size of the disk in the system, and a search list of which disk to install on.

To show an example we will describe how to convert a standard SuSE Linux boot disk into a boot disk for automatic installation. A standard SuSE Linux boot disk contains the following files (date and size will obviously vary, depending on the SuSE Linux version and boot disk used); see Listing 1.

The file "linux" is the kernel image that needs could be replaced with a kernel image that contains the necessary drivers for the hardware installed in the target system. Alternatively it is possible to use the same module loading technology that is available during a standard interactive install.

Simply add an insmod rule to load modules. This rule is also part of the info file on the boot media, i.e.,

```
insmod tulip
insmod ncr53c8xx
insmod ne irq=10
```

These lines load the tulip, nec scsi and a ne2000 driver. The last line is an example how to give parameters to the module. Obviously, insmod is used in the info file in a very similar way as on the command line.

The file "syslinux.cfg" contains the boot parameters and needs to be slightly modified in order to start the automated boot process. The additional parameter "linuxrc=auto" needs to be added to the "append" line, so that it looks similar to

```
append ramdisk_size=10240
  initrd=initdisk.gz rw
  linuxrc=auto 2
```

Now the control file can be added to the boot disk as /suse/setup/descr/info. This control file contains most of the machine specific information that is necessary to install SuSE Linux without administrator interaction.

There is also an option serial x [y] which tells the kernel to connect to the console via the serial interface dev ttyS *x* with baud rate *y*. Of course the kernel on the disk must be configured to support a serial console. This option is very useful if you want to control the setup of servers remotely, which is often a requirement at large installations. This allows to use a terminal server for this purpose.

```
-rwxr-xr-x   1 root     root       318977 Apr 14 15:10 initdisk.gz
-r-xr-xr-x   1 root     root         5480 Mar 24 11:56 ldlinux.sys
-rwxr-xr-x   1 root     root       658155 Apr 14 15:10 linux
-rwxr-xr-x   1 root     root         2424 Apr 14 15:10 message
-rwxr-xr-x   1 root     root       102665 Apr 14 15:11 net-mod.gz
-rwxr-xr-x   1 root     root       367500 Apr 14 15:11 scsi-mod.gz
-rwxr-xr-x   1 root     root          107 Apr 14 15:10 syslinux.cfg
```

Listing 1: Boot disk contents.

IP Definition and Installation Source

All settings for the "/etc/rc.config" file can be given there, additionally several others that are normally interactively requested during the install process. A short example should serve as an illustration of the possibilities.

```
# first tell the basics and
# where to install from
Language:    English
Display:     Color
Keytable:    us
Bootmode:    net
Netdevice:   eth0
Ip:          192.168.1.15
Netmask:     255.255.255.0
Server:      192.168.1.1
ServerDir:   /install/SuSE62
Nameserver:  192.168.1.1
RC_CONFIG_0 FQHOSTNAME
          host.your.domain.org
```

This control file will create a system called host.your.domain.org with IP address 192.168.1.15 installed from the server 192.168.1.1, directory /install/SuSE62, using the name server 192.168.1.1.

Of course it is also possible to obtain the information on IP address, netmask, default gateway, nameserver, install server and install directory on that server using DHCP. The fact that DHCP is not necessary for the automated install is what makes this so flexible.

```
AUTO_NET        1
```

The AUTO_NET rule tells YaST to configure the network for the installed system in the same as it was configured during the installation. This works for both static and dynamic IP addresses.

If a machine is configured using DHCP it is possible to obtain the correct nameserver (and therefore, through reverse lookup, hostname) from the information provided by the DHCP server. This is enabled through the following settings:

```
AUTO_NAMESERVER     1
AUTO_NAME           1
```

Setup Of System Parameters and Applications

The automated installation of a SuSE Linux system using YaST includes both, the installation of the system, system programs and applications and the configuration of the system, its components and applications. The automated configuration uses the interfaces which come with SuSE Linux. Most of the configuration can be done by setting up /etc/rc.config:

```
# next some settings for
# the resulting client
RC_CONFIG_0 MOUSE /dev/psaux
RC_CONFIG_0 GPM_PARAM -t ps2 \
             -m /dev/mouse
RC_CONFIG_0 START_GPM yes
#
RC_CONFIG_0 ROOT_LOGIN_REMOTE no
RC_CONFIG_0 MAX_DAYS_FOR_LOG_FILES 365
RC_CONFIG_0 CONSOLE_SHUTDOWN reboot
#
RC_CONFIG_0 TIMEZONE EDT
RC_CONFIG_0 START_INETD yes
RC_CONFIG_0 SMTP yes
RC_CONFIG_0 START_ROUTED no
RC_CONFIG_0 START_NAMED no
RC_CONFIG_0 MODEM
RC_CONFIG_0 SENDMAIL_TYPE no
```

This system has a PS/2 mouse attached and runs gpm. It will not accept remote root logins (except through ssh), will hold the central log files at most 365 days and react to pressing CTRL-ALT-DEL on the console with a reboot of the machine (and so on).

So in order to configure the majority of the applications available, all you have to do is add RC_CONFIG_0 rules to the info file. This is an easy, straight forward way to setup the system **and** the applications installed. Additionally, future extensions or new parameters (i.e., if customer specific applications require additional variables) can be implemented using this rule.

Drives, Partitions, and File Systems

Of course, this is missing one crucial piece of information. The disk layout of the system. There are several ways to provide that information. The standard case that we want to discuss here assumes a hands-off installation, therefore there will be no interaction asking the user whether he wants his disk partitioned and whether he likes the layout. To indicate that the setting

```
AUTO_FDISK 2
```

would be used in the control file. A value of 0 would indicate that the disks shouldn't be partitioned at all (that might make sense if the disk is already partitioned, for example) and a value of 1 is used to do the automatic partitioning, but to display the layout to the user for possible change and approval.

The interesting question is of course which disks to partition. Those are given as a list in the variable

```
AUTO_FDISK_DISK /dev/sda /dev/hda
NO_ASK_SWAP     1
```

which would indicate to install on the first SCSI disk, and if that doesn't exist, on the first IDE disk. The partition layout for the install disk is given on the install media in a file in the directory "suse/setup/descr". The file name is "part_NNNNN", where NNNNN is a five digit value that indicates the minimum number of megabytes that the disk must have for this scheme to be applicable. If, for example, two files "part_00000" and "part_02500" exist, then the former is used for all disks with less than 2500 megabytes, and the later is used for all disks with 2500

megabytes and more. So with this rule and the size based partition file selection we have a very flexible and automatic default disk layout.

Alternatively, you can use AUTO_FDISK_DISK to hold multiple (nearly) independent SuSE Linux systems on one machine, each system on one disk, i.e., to test beta versions. The only point of conflict is the Linux Loader /lilo in the master boot record of the "first" disk.

If the type and size of the installed disks is known in advance, a fixed assignment between partition information and disk is possible as well. The setting

```
AUTO_FDISK_TABLE /dev/sda
    $I:/suse/ADD_FILES/part_diska
AUTO_FDISK_TABLE /dev/sdb
                        $D:/part_sdb
```

will get the partition information for the first and second SCSI disk from the files specified on the install media. $I: is substituted with the mount point of the install media which can be a physical connected drive or a NFS mounted filesystem. $D: stands for the mount point of the boot filesystem. So the second rule says that YaST can find the partition information for the second SCSI disk on the boot disk.

This method is very flexible and can handle any layout from a single disk workstation up to a complex file server with several disks, even on different buses.

Setting NO_ASK_SWAP to 1 again ensures a hands-off installation, as the swap settings from the partition description are used. An example for such a description could be

```
/      200
SWAP   100
/usr   800
NONE     20
NONE     30
/home  0
```

which would create /, /usr and swap with the at least the sizes given (cylinder boundaries are used as partition boundaries, therefore the sizes are rounded up somewhat) and assign the rest of the disk to the /home partition.

You can also specify FSTAB_SEARCH /dev/sda1 /dev/hda1 to search for an already existent file system table (/etc/fstab) which defines the assignment of devices to mount points.

YaST reads the first /etc/fstab and takes the layout given there. In that case the disk's partition table will not be changed. The already existing partitions will be used instead. If no /etc/ftab is found, auto partitioning is started as defined in the AUTO_FDISK* rules.

The rule FSTAB_FORMAT defines a list of filesystems, which have to be formatted before the installation starts. Of course, each new partition (except of the partitions with NONE as mount point) is formatted. This feature, in combination with FSTAB_SEARCH, can be used to format the /, /var and /usr file systems but to leave the data in /home, /space or /databases alone.

The combination of these options shows that automatic installation and configuration is not only an aspect of building up a completely new system, but is also a resourceful way to update a system or to (partly) rebuild it after a complete server crash.

Selection of Packages to be Installed

The selection of packages to install is the final point in creating the configuration necessary for an automated install. Here standard selection files as used by the SuSE setup tool "YaST" can be used. This makes creating of such selection files very simple, as it can be done interactively using YaST. The location of such files can be given with

```
FAST_INSTALL 2
AUTO_INSTALL default.sel
ADD_INSTALL  additional.sel
AUTO_KERNEL  scsi01
```

The first line ensures that there is no prompt that asks the user whether he wants a hands-off, automated install.

The selection files themselves are simply the files that YaST stores when asked to save a configuration that was interactively created using YaST. Of course it is possible to edit these files in order to install customized RPMs. Similarly, running shell scripts during the install is possible as well. Additionally, custom scripts can be executed at install time. This allows the system administrator full control over the configuration of the system that is being installed.

The last line in the example above identifies the kernel that should be used. This kernel must include the drivers for the hardware that the system needs for booting, especially the SCSI drivers, if necessary. All other drivers can then be loaded as modules, as in any standard Linux installation.

Custom Scripts, Benchmark And Principles

Even though the majority of the tasks needed for automated installation can be solved with the features described so far, the automated installation using YaST offers yet another additional configuration interface; see Listing 2.

```
PRE_SCRIPT $I:/suse/ADD_FILES/bin/prepare_something
POST_SCRIPT $I:/suse/ADD_FILES/bin/configure_own_apps
```

Listing 2: Custom configuration.

The first entry point to run custom configuration scripts is PRE_SCRIPT. All scripts, which are added to the pre_list are called before any package is installed. The second entry point is POST_SCRIPT. The scripts in the post_list are called after the last package of the first installation media is installed. On network based installations no media change is needed during the installation. But if you use CDRom you can use LAST_SCRIPT to define scripts that are execute after the installation of the last package. Media changes are not done automatically, therefore an installation media which contains all the installation packages is to be preferred.

While no extensive benchmarks of the automatic installation process have been performed, some initial tests seem to indicate that this is a very fast way to install systems. For example, for a small mail server system containing all packages necessary for this task, from the moment of first accessing the boot disk to the final login prompt only about 5 minutes elapse.

In order to obtain an as homogeneous environment as possible, it is usually preferable to do some planning and some generalizing on the types of systems that will be installed. If, for example, the systems that are to be installed, are used as front end systems in a call center, then the disk layout and the selection of packages installed will usually be very similar. Creating a flexible template for all these systems allows the system administrator a very fast and easy configuration of the boot disk for such a system with minimum administrative overhead. If the hardware is similar enough (same type of disk interface, identical network cards or one of just a few specific network cards in each system) then even a single boot disk can used for every system that needs to be installed.

Special systems (like central servers or gateways) can still be configured without any additional overhead, since the generic defaults in the template can always be overridden with a special configuration file on the boot disk. By providing classes of systems and putting each system in such a class it is possible to have a single boot disk image and a single install server (or a network of identical, redundant install servers) to install all Linux systems at a site.

Another very important use for the automated install is the concept of clone backups. This allows to create a backup tape that contains a complete dump of all file systems of a machine, plus a boot disk that automatically restores the contents of such a backup tape. This allows easy to use disaster recovery provisions, as all configuration information, the partition layout and all information about the software, drivers and tools, that are installed on the system, is maintained and can be restored in a single step.

On systems with local data storage (home directories, data base table and index spaces, web server pages) the backup process can similarly be divided into two steps. The system backup that restores the system itself and the software that is installed on it, and the data backup that contains the variable data on the system. The system backup needs to be done less frequently, mostly triggered by changes to the configuration. The data backup on the other hand is now smaller and can be done according to a regular schedule.

Conclusion

The methods to install and restore a system presented in this paper allow administrators of small and large sites to automate a large part of the installation and recovery process for SuSE Linux machines. This provides an important piece of making Linux ready for the requirements in commercial environments. The methods described are flexible enough to allow a broad type of uses, depending on the environment the systems are installed into. By using only documented interfaces of the SuSE Linux install process, these procedures can be used not only with the current version, but also with future versions of SuSE Linux, which makes the investment in such an installation environment worth while.

Author Information

Dirk Hohndel is Vice President of Strategic Development of SuSE GmbH and CEO of SuSE Rhein/Main AG, a subsidiary of SuSE that provides professional services and consulting. He also serves as Vice President of The XFree86 Project, Inc., a non-profit corporation that provides an Open Source implementation of the X Window System on PC Unix systems like Linux.

Prior to his current position at SuSE, Dirk was Unix Architect at Deutsche Bank AG, before that he worked as Senior Software Engineer for AIB Software Corporation (which was then acquired by PLATINUM technology).

Dirk holds a Diploma in Mathematics and Computer Science from the University of Würzburg, Germany. He is married and lives near Frankfurt, Germany.

Fabian Herschel works as Senior Consultant at SuSE Rhein/Main AG. His projects currently include automated installation procedures for Linux systems in a business critical customer environment.

Before this, Fabian was Unix Specialist at Deutsche Bank AG. He had the central responsibility for the design of HACMP clusters and the evaluation of storage systems for a data center. Prior to that he was System Consultant for SerCon GmbH a subsidiary of IBM. He was responsible for SAP R/3 systems and implemented a data center automation for multiple decentral R/3 systems.

Fabian holds a Diploma in Computer Science from the University of Darmstadt, Germany. He is married and lives near Mainz, Germany.

Bibliography

[1] http://www.suse.de/linux/whitepapers/yast.

[2] http://wwwwswest2.sun.com/smcc/solaris-migration/cookbook/jump1.html.

[3] http://www.hp.com/esy/software_applications/systems_management/products/ignite_ux_wp.html.

[4] http://www.rs6000.ibm.com/doc_link/en_US/a_doc_lib/aixins/aixnimgd/toc.htm.

[5] Branchstart – A Generic, Multi-OS Installation Server. Rory Toma, WebTV Networks Inc., LISA 98.

[6] http://wwwcache.ja.net/dev/kickstart.

Enterprise Rollouts with JumpStart

Jason Heiss – Collective Technologies

ABSTRACT

JumpStart is Sun's solution for installing the Solaris operating system. The Custom JumpStart feature allows the installation process to be automated. However, configuring boot and NFS servers and the appropriate name services for JumpStart is a time-consuming and error-prone process. The scripts that Sun provides do not help this process much. This paper will talk about how to better automate the configuration steps needed to perform JumpStarts over the network, emphasizing speed and accuracy as well as simplicity from an operator's point of view. The infrastructure needed to perform large numbers of simultaneous JumpStarts is also discussed. By automating the actual rollout process, it is possible to JumpStart several hundred machines at once. Techniques for doing so are presented. The improvements presented in this paper allowed one company to improve the speed of their rollouts to 600 Sun workstations by more than an order of magnitude. This greatly decreased user downtime and saved the company hundreds of thousands of dollars. Lastly, possible future improvements to the process are discussed.

Introduction

JumpStart is Sun's solution for installing the Solaris operating system. JumpStart has three modes. The Interactive and Web Start modes are for interactive installs. While fine for installing Solaris on a single machine, this clearly doesn't scale to the enterprise level. The third mode is what Sun calls Custom JumpStart, which allows the install process to be automated. The information in this paper is based on the Sparc architecture. Solaris and JumpStart are available for the Intel architecture but the limitations of that architecture make large-scale rollouts difficult. Specifically, booting a PC off of the network is generally not possible. Instead, booting off of a floppy is usually required. This requires generating hundreds of boot floppies and visiting each machine to insert one.

Configuring a Custom JumpStart is already well covered in Sun's book *Automating Solaris Installations* [Kas95], with Asim Zuberi's article "Jumpstart in a nutshell" [Zub99] presenting some useful updates, as well as Sun's documentation in the Solaris Advanced Installation Guide [SAIG]; thus those topics will not be covered here. Instead, this paper will talk about how to automate rollouts using JumpStart. While Custom JumpStart makes it possible to automate the process of installing Solaris (partitioning disks, installing packages, etc.), it is still a manual process to configure boot and NFS servers and name services so that the workstations can boot off of the network and start the Custom JumpStart. Automating that configuration process and performing the rollout itself are the topics of this paper. The infrastructure required for large-scale rollouts will also be discussed.

Interactive Installs

There are a number of problems with interactive installs (either Interactive or Web Start) on an enterprise scale. First, they are very laborious. Interactive installs take at least one hour per machine. While good for system administrator job security, this is expensive and tiresome. It also makes it difficult to get timely upgrades to the users. Second, with interactive installs it is tempting to customize the install for each user. This inconsistency in the installs makes future maintenance more difficult. A little more design and research work up front will usually allow an administrator to develop a standard installation that meets the needs of all users.

Custom JumpStart

Let's review how a Custom JumpStart works. Custom JumpStart makes it possible to automate the entire installation process. A Custom JumpStart begins with the client workstation booting off of the network in exactly the same way as a diskless workstation.

The first thing the client needs to do is get its IP address. This allows it to send IP packets onto the local subnet. The client gets its IP address using a protocol called RARP, or Reverse Address Resolution Protocol. The client knows its ethernet address but not its IP address. So it broadcasts a packet out on to the subnet that says, "My ethernet address is xx:xx:xx:xx:xx:xx, does anyone know my IP address?" With Solaris (and most other operating systems), you need to be running a special server to listen for and respond to RARP packets. This server is called in.rarpd in Solaris. The server listens for RARP packets, and when it receives one it attempts to lookup the ethernet address in /etc/ethers or the ethers NIS/NIS+ map (depending on the ethers setting in /etc/nsswitch.conf). If that is successful, the server gets the hostname for the machine. Next, the server attempts to lookup the IP address for that hostname. If that is successful, the server responds to the client with a packet saying "Hello xx:xx:xx:xx:xx:xx, your IP address is xxx.xxx.xxx.xxx." Note: the presence of a /tftpboot directory causes the standard Solaris

initialization scripts to run in.rarpd and rpc.bootparamd (more on bootparams later).

The client now has its IP address and the next thing it needs is the inetboot boot block. This is a very basic kernel that knows how to do enough IP networking to NFS mount a root directory and load the real Solaris kernel. The client downloads the inetboot file using a protocol called TFTP, or Trivial File Transfer Protocol. It has no authentication or security and very basic error handling. This makes it easy to implement TFTP in the limited memory of the Sun PROM but also makes the protocol very slow and a security risk. Speed, however, is not an issue with the size of the inetboot file (~150 kB). The security implications of TFTP will need to be handled in a way that complies with local security policy. In general, hosts don't run TFTP servers. Under Solaris, the TFTP server is started by inetd. By default, the entry in /etc/inetd.conf for in.tftpd, the Solaris TFTP server, is commented out.

Once the inetboot file has been transferred, the client executes it. The first thing that the inetboot file does is to send out a bootparams "whoami" request. bootparams is a protocol that allows the transfer of key:value pairs from a server to the client. The client sends out a request for the value associated with a key and the server responds with that value. The bootparams server under Solaris is called rpc.bootparamd. The server listens for bootparams requests and when it receives one it tries to lookup the hostname corresponding to the IP address in the request. If that succeeds then the server tries to look for an entry matching that hostname in either /etc/bootparams or the bootparams NIS map (depending on the bootparams entry in /etc/nsswitch.conf). If that succeeds then the server sends a response packet back to the client with the value for the key that the client requested. A "whoami" request is special, however. The bootparams server responds to a "whoami" request with the client's hostname, NIS/NIS+ domainname and default router. The client next sends out an ICMP address mask request. Using the netmask from the ICMP request and the default router from the bootparams request, the client finishes configuring its network interface and routing.

The client then makes a bootparams request for the root key. The server returns the NFS path for the client's root directory and the client performs an NFS mount of that directory. At this point, the client loads the regular Solaris kernel. The JumpStart root directory and startup scripts are different from the standard Solaris configuration so the rest of this process differs from a normal diskless boot process. The startup scripts repeat all of the network configuration steps that the inetboot boot block performed. They then NFS mount the install directory, which is a copy of the Solaris media, at /cdrom. The path to the install directory is found via a bootparams request for the install key.

The client then NFS mounts the profile directory, found via bootparams using the install_config key. This directory contains the rules and profiles as well as the begin and finish scripts for a Custom JumpStart configuration. At this point the JumpStart scripts collect a few bits of information, like the local timezone and locale, from NIS or NIS+. Starting with Solaris 2.6, this information can also be configured through a local file. The rules file is then searched for a set of rules that matches the client's hardware. If a matching rule is found then a corresponding begin script, profile and finish script are selected. The begin script, which is run before anything on disk is changed, is typically used to save things like local calendar files or crontabs. The profile dictates how the disks in the client are partitioned and which Sun package cluster is installed. The clusters range from the Core cluster, which is the minimum software needed to boot Solaris, up to the Entire Distribution cluster. Typically the End User or Developer clusters are selected, depending on whether support for compiling programs will be needed. The packages associated with the selected cluster are installed once disk partitioning is completed. Then Maintenance Update patches are installed if available. Maintenance Update patches are what differentiate an FCS (First Customer Ship, i.e., the first release shipped to customers) release of Solaris from a later dated release of the same version. Once the Maintenance Update patches are installed, the finish script is executed. This is typically where site-specific customizations would be located. One very common finish script task is to install the latest Recommended Patch Cluster from Sun. Once the finish script exits, the client reboots and the JumpStart is complete.

The network booting and JumpStart processes require some network and system infrastructure in order to work. The network boot process primarily depends on a "boot" server. The clients need a server on their subnet to make requests to until they possess enough information to properly handle routing. As such, a boot server must be connected to each subnet. A boot server would run RARP, bootparams and TFTP servers as well as being a NIS slave in a NIS environment. It is possible to configure JumpStart to handle an environment without NIS+ replicas on each subnet. The other important piece of infrastructure is the NFS server. We will discuss how to size NFS servers for JumpStart later in this paper.

Figure 1 shows an example JumpStart environment. The NFS server, which serves as a boot server as well, is located on the engineering subnet. Marketing is on their own subnet so they need a separate boot server. They will then use the same NFS server as the engineering group.

The Old Way

When I first started working with JumpStart, the company I was working at had two employees

assigned to performing Solaris updates on deployed workstations. The process at that time was manual and significantly error-prone. JumpStart configuration was done using the add_install_client script provided by Sun. On days when a JumpStart was scheduled, the two operators would attempt 10-20 JumpStarts during the lunch hour. On average, the JumpStarts failed on 30% of the machines, usually due to errors in the NIS maps or hardware that didn't match any of the JumpStart rules.

Given the high failure rate, the process generated much distrust and resentment in the user community. Managers knew that this was going to cost them downtime and did their best to avoid having their users' machines upgraded. The process was costing the company a lot of money. After the second day of watching this, I realized several things:

- People make a large number of typos. Booting a machine off of the network requires a number of files and NIS/NIS+ maps to be configured exactly right. A single error will cause the entire JumpStart to fail in various, difficult to diagnose, ways.
- Users are greatly annoyed when they come back from lunch and their machines are not functional. The 30% failure rate resulted in this happening to a number of users. Each failure typically took at least a few minutes to diagnose. Since the JumpStart itself took at least an hour, if anything went wrong the user's machine was down during part of their normal working hours. This frustrated the users and cost the company money.
- The whole process was mind-numbingly slow. There was a lot of repetitive typing that went into configuring each machine using the add_install_client script. Additionally, there was quite a bit of manual error checking to do because add_install_client does only limited error checking. Further, visiting each machine to start the JumpStart took a long time because they were frequently spread out over a large area.
- The mix of NIS maps and local name service files created a great deal of confusion and problems. add_install_client checks the appropriate NIS map for entries but if it doesn't find one then it adds the entry to the local name service file on the boot server. This meant that the operator had to check both NIS and the local files when searching for missing or incorrect entries.
- The process didn't get the OS upgrades to users in a timely fashion. When I started working with the two employees, they were just finishing up a rollout of an updated version of the JumpStart image. It had taken them close to six months to do this for the roughly 600 standard Sun workstations in use. If a serious bug had been found in the image, it would have taken another six months to roll out a new version. The process in use clearly was not scaling to the number of machines to which it was being applied.

So, I set about to devise a better mousetrap. My goals for the new process:

- The process for configuring the JumpStart boot process should be quick and require minimal typing.
- Far more complete error checking should be done in order to reduce or eliminate JumpStart failures.
- The scripts developed should fully support the use of NIS or NIS+ and not create entries in local name service files.

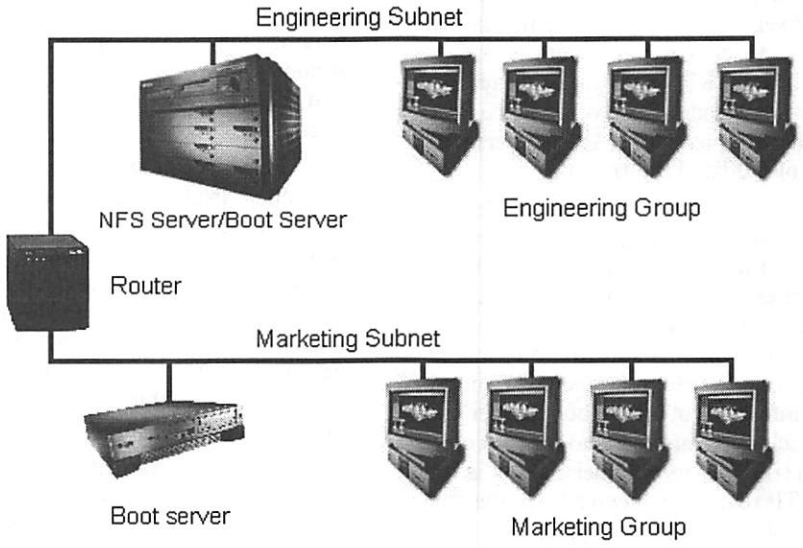


Figure 1: Example JumpStart environment.

- The process should scale to hundreds or even thousands of machines.
- A reliable method for kicking off the JumpStarts without physically touching the machine should be developed.

The New Way

Over the course of the next several rollouts at this company, I was able to develop scripts and processes that allowed me to accomplish all of my goals. At the time I left we were able to JumpStart 200 machines in an hour. A complete rollout of a new image to all 600-650 machines could be completed in about 10 hours over the course of several evenings. This was a great improvement over the six months it used to take to perform a rollout.

Config

The first thing I did was to develop a replacement for add_install_client. I called my script config. Given the complexity of what I wanted to do, I didn't think the Bourne shell was the right tool for the job. So, I decided to write config in Perl.

The biggest advantage of config over add_install_client is that it checks for almost every problem that could possibly prevent a JumpStart from working. This includes errors that prevent the machine from correctly booting via the network and errors that would cause the Custom JumpStart to fail.

There are several things that can go wrong with the first step, the RARP request. First, the client's ethernet address must be correctly mapped to its hostname in the ethers map. There should only be one entry for each ethernet address. Each hostname needs to be in the hosts map correctly mapped to the IP address assigned to that hostname. Again, there should only be one entry for each hostname/IP address pair. Also, the boot server needs to be running a RARP server and no other host on that subnet should be running another RARP server. The reason for this is that multiple replies tend to confuse the client. Also, if there is more than one RARP server on a subnet, the other servers are typically ones that we don't control. People have a bad habit of leaving RARP servers running with incorrect or outdated information.

The client then tries to TFTP the inetboot file. For this to happen, the boot server must be configured to run a TFTP server and there needs to be a boot block in /tftpboot that matches the IP address of the client in hexadecimal. The way this is usually done is that one copy of the inetboot file, which is specific to each version of Solaris and each kernel architecture (sun4m, sun4u, etc.), is put into /tftpboot on the boot server and then symlinks for each client are made to the appropriate inetboot file. The TFTP server under Solaris is run by inetd when a TFTP request is received; so, the tftpd entry in /etc/inetd.conf must be uncommented. inetd must be sent a HUP signal if a change is made to inetd.conf. Otherwise, inetd won't re-read the configuration file. Here, again, we only want one server per subnet. If the machine running the RARP server doesn't respond quickly enough to the client then the client will broadcast any further requests. That means that if there is more than one TFTP server on that subnet with the appropriately named file, they will all respond. As with RARP, multiple TFTP replies tend to confuse the client and it will usually hang. Also, we want to avoid rogue servers with incorrect or outdated inetboot files.

The client then sends out a bootparams request. This, again, requires that the client's IP address be correctly associated with its hostname in the hosts map. Also, there must be an entry matching that hostname in the bootparams map. The boot server must be running a bootparams server (rpc.bootparamd in Solaris). Also, like with RARP and TFTP servers, there should only be one bootparams server on the subnet. Multiple replies don't cause as many problems as they do with RARP or TFTP; but, again, we want to avoid rogue servers with incorrect or outdated information.

The client then performs an NFS mount of its root directory. This requires that the root directory be located on the fileserver and that the directory is exported with the options of ro,anon=0. Exporting the directory read-only isn't required but it is a very good idea. (At various times, there have been bugs in Sun's JumpStart scripts such that things will break if the root directory is exported read-write. This provides even more reason to export it read-only.)

The client then performs NFS mounts of the CDROM image and the profile directory. This requires that the fileserver exports the CDROM image and the profile directories, with the same options as the root directory.

In an NIS environment, the client will then broadcast a request for an NIS server. This is the default behavior in an NIS+ environment as well, but it is possible to add an extra field to the client's bootparams entry to point it at an NIS+ server on another subnet. Thus, in an NIS environment it is necessary that there be an NIS server on the same subnet as the client.

The last thing that can go wrong is with the JumpStart itself. One of the first things that happens during a JumpStart is that the machine checks its hardware against what is called a rules file. The rules file defines various hardware configurations and what profiles go with them. The profiles dictate how the disks in the client are to be partitioned, as well as the Sun package cluster to be installed. In order for this to succeed, the machine's hardware must match at least one of the configurations in the rules file.

config requires root access to the NIS master in a NIS environment. This is most easily accomplished by running config as root on the NIS master, and that is what we did. However, site configuration and security policies might dictate using some other method. This access allows config to automatically make NIS map

corrections and push the updated maps. Otherwise, the operator is responsible for making the map changes manually. Removing dependencies on the operator typing information correctly is one of the main reasons for developing config. config is very careful about how it makes changes and we never experienced any corrupted NIS maps due to its operation.

add_install_client does not have a built-in option for bulk configuration. Automating bulk configuration with add_install_client is difficult because it requires knowledge of the client kernel architecture for each machine. With config, I built in support for bulk configuration. The script takes a list of hostnames and attempts to configure all of the machines on the list. config queries each machine for the information it needs, including the kernel architecture. This requires root login access on all of the clients. The operator then need only create the list of hostnames.

In combination with the scripts mentioned below, config tracks the status of each machine. This is especially useful when using the bulk configuration feature described above. The operator compiles a list of several hundred hostnames and runs them through config. The script will print a report at the end of its run listing which machines failed and the reason for the failure. The operator can then fix the problems and rerun config with the same list of hostnames. config knows which machines are already configured and skips over them. This saves hours of waiting when working with a large number of machines. This status tracking is accomplished through the use of a text file listing each machine and a numeric code indicating what state the machine is in.

Another advantage of config is that it automatically picks the boot and NFS servers for a machine when configuring it. This allows the operator to be blissfully ignorant of the network topology and the various servers. With add_install_client, the operator has to know which boot server is on the same subnet as the machine. While usually easy to figure out, it is one more mistake that can be made. The logic that config currently uses to pick the NFS server is simplistic, but could be improved to use traceroute to truly find the closest machine. Currently config looks for an NFS server on the same subnet as the client, if it doesn't find one then it picks a default server. This works well if most clients are on a subnet with an NFS server but would fail in an environment where this isn't the case.

config creates a centralized place to make JumpStart configurations. With add_install_client in a large environment, the operator must know which boot server to use for each machine and log into that server to perform JumpStart configuration. With config, the operator always logs into the same server and doesn't need to know which boot server to use. This also prevents uninformed operators from setting up more than one boot server on a subnet.

With config, a method for users and administrators to tag machines as non-standard is introduced. Machines are tagged as non-standard by creating /usr/local/do_not_jumpstart. These were usually machines that had been standard desktop clients but then additional software or other modifications were made to the machine. We didn't want to JumpStart these machines because that software would have been lost. Tagged machines are automatically skipped over, eliminating the need for a failure-prone, manual method of keeping track of these machines.

As you can see there are many advantages to replacing add_install_client with a more capable script. Hopefully Sun will realize this and improve add_install_client.

Infrastructure

The network and server infrastructure becomes important when JumpStarting many machines at once. When installing 10 machines at a time, the CD image can be kept on a single drive on a Sparc 2 at the end of a shared 10 Mbit Ethernet connection. However, scaling up to JumpStarting several hundred machines at a time requires more bandwidth.

The first consideration is boot servers. The clients need a server on their subnet to make requests to until they possess enough information to properly handle routing. As such, a "boot" server must be connected to each subnet. Almost any sort of machine will work as a boot server since the total amount of data transferred from the boot server to each client is approximately 150 KB. Note that add_install_client requires the JumpStart root image to be located on each boot server. config does not require this and thus the bandwidth is limited to the inetboot file, which is approximately 150 KB. If you need or choose to be compatible with add_install_client then you will need to take into account the additional disk space and bandwidth requirements of the root image. This image is approximately 30 MB on Solaris 2.5.1 and earlier, it jumped to approximately 150 MB in Solaris 2.6 and later versions. If your network design already contains a server on each subnet with Sun clients then boot servers are not a concern. If not, then you will need to put something together. We used several Sparc 5s with quad Ethernet cards. Doing so allowed a single machine to serve up to five subnets (including the built-in interface in the Sparc 5).

There is one very subtle trick to using multihomed boot servers. I mentioned earlier that the client gets its default router via a bootparams "whoami" request. The boot server, like most machines, would normally be configured with only one default router as that is all it typically needs. This default router is what the boot server would then send in response to a "whoami" request. The problem is that on a multihomed boot server, that default router will be correct only for clients on the same subnet as that default router. The solution is to put multiple entries in /etc/defaultrouter, one for the default router on each subnet that the boot server is attached to. Then, when the

boot server boots, it will create multiple default route entries in its routing table and will return the correct one, based on the subnet, in response to "whoami" requests. It is also generally a good idea to touch /etc/notrouter on multi-homed boot servers so that they don't route packets between the various interfaces.

The next consideration is network bandwidth. As a rough estimate, each workstation will transfer about 500 MB in the course of the JumpStart. The client's network connection is of little concern as 10 Mbit Ethernet can easily handle that much data in an hour. However, the bandwidth of the NFS servers' network connections must be considered. It is easy to figure out the required network bandwidth. As an example, say you want to JumpStart 200 machines in one hour. 200 times 500 MB divided by 1 hour is 100 GB/hr. Network bandwidth is usually measured in Mb/sec. By a simple conversion (1 GB/hr =~ 2.2 Mb/sec) we see the required bandwidth is about 220 Mb/sec. Switched 100-Base-T is fairly common these days and, as the name implies, runs at 100 Mb/sec. If you can spread the load evenly over three switched 100-Base-T interfaces or three servers with 100-Base-T interfaces then there will be sufficient bandwidth. Keep in mind that the network traffic of JumpStarting many machines will cause a high collision rate on shared Ethernet, so avoid it if possible. Server interfaces on shared Ethernet will only produce about 10% (at best) of the rated bandwidth during JumpStarts.

The last issue is NFS server capacity. The most common measure of server capacity for NFS servers is the SPEC SFS benchmark. SFS93 was also known as LADDIS, the current version of SFS is SFS97. The results of the test are expressed as the number of NFS operations (NFSops) per second the server can handle. Results from SFS93 and SFS97 are not comparable. Brian Wong, of Sun Microsystems, estimates that each client requires 50-70 NFSops/sec (as measured by SFS93) during a JumpStart [Won97]. This matches very closely with the performance we observed as well. SPEC provides SFS93 and SFS97 benchmark results for a wide variety of NFS servers on their website at http://www.specbench.org/osg/sfs97/. Sun also provides results from the older LADDIS benchmark on their website. For example, a Sun Enterprise 3500 with four CPUs can handle approximately 8900 SFS93 NFSops/sec according to Sun. At 60 NFSops/sec/client, that server could serve approximately 150 clients. To achieve this level of performance, the JumpStart data should be striped over a number of disks using RAID 0 or 5.

Start

The last step was to develop a script to start the JumpStart process on a large number of systems simultaneously. The simplest form of such a script would loop through and rsh to each machine in order to execute reboot net – install". Unfortunately, it takes approximately 15 seconds to make the connection and initiate the reboot for each machine. When attempting to start 250 machines at the same time, it will take more than an hour just to get all of the machines started. Additionally, a few things should be checked on the machine first, such as whether there are any users logged on. So now, the naive implementation of a kickoff script will take a couple of hours just to get all of the machines started. The start script I developed forks into a number of processes, each of which is assigned a group of hosts to work on. This allows the operator to get all of the machines started within a few minutes of each other. start also references the status file updated by config to ensure that it only starts machines that were successfully configured. The script checks to see if any users are logged in. It also checks to see if the machine has been tagged as non-standard. Although config checks for this as well, it is possible that someone might tag the machine in the day or two after config is run but before the rollout is performed. start also checks to see if the root password is the standard root password. The clients will typically all have a common root password, but sometimes when someone customizes a machine he will change the root password. This was put in as a last-ditch effort to catch non-standard machines even if they had not been tagged as non-standard.

Once start has started the install on a machine, it continues to attempt to rsh in periodically and check on the progress of the JumpStart. This progress is recorded in the status file. This requires a minor change to /sbin/sysconfig in the JumpStart root image on the NFS servers to start inetd, and thus allow rshd to start, during the JumpStart.

During a rollout we would have several people on hand to deal with any problems that arose during the JumpStart. To help everyone spot problems as quickly as possible, I wrote a CGI script that references the same status file used by config and start. The script generates web pages with the status of all of the machines indicated by red, yellow or green dots and a status message. The pages automatically refresh in order to display up-to-date information. As machines encounter problems, they are flagged on the web page with yellow or red dots depending on the severity of the problem. The people on hand then read the status field and determine if the problem needs intervention. Most common were users who remained logged in during the scheduled rollout. In those cases, someone would call the user or visit the machine and ask the user to log out. Other problems were similarly handled in order to keep the rollout moving. This allowed the client to scale up to performing several hundred simultaneous JumpStarts. We had several people on hand to deal with the rare problems that cropped up. They would watch the web pages for troubled machines, figure out what the problem was and attempt to fix it. Our record was just over 250 machines in an hour and a half.

QuickJump

As mentioned previously, a standard Custom JumpStart consists of a begin script, disk partitioning,

Solaris package installation, possible Maintenance Update patch installation and then typically additional patch installation during the finish script. Package and patch installation on older machines, like a Sparc 5, can be quite slow. We found that installing the Recommended Patch Cluster for Solaris 2.5.1 would take eight hours or longer on a Sparc 5. This in turn makes the JumpStart take a long time. Mike Diaz of Sun Professional Services developed a system for this client we called QuickJump to speed up the JumpStart process. Rodney Rutherford and Kurt Hayes of Collective Technologies further refined this process.

QuickJump consists of performing a standard Custom JumpStart once on a machine in the test lab. Once the JumpStart completes, each filesystem on the client is dumped using ufsdump to the NFS file server. Then we create a separate Custom JumpStart configuration for use in the rollout. In that configuration we select the Core cluster of Solaris packages, which is quite small, in the JumpStart profile. We also disable Maintenance Update patches if the Solaris version in use has them [SI15834]. Then we modify the finish script in the JumpStart configuration to restore the ufsdump images onto the client. This overwrites everything on disk, namely the Core cluster of Solaris packages. For even greater speed, it is possible to disable the installation of a package cluster altogether, however this requires somewhat extensive modifications to the Sun JumpStart scripts. The Core cluster takes only a few minutes to install so we decided that it wasn't worth the effort to make those modifications. The finish script then merely has to change the hostname and IP address as appropriate and it is done. This vastly speeds up the JumpStart process on older hardware, bringing it down to 45 minutes or so for a Sparc 5. The difference becomes negligible with machines based on the UltraSPARC processor, so QuickJump may not be necessary once the older hardware is replaced.

The Future

It has now been a year since I worked on the project. Although the client is still successfully using these scripts and processes largely unchanged, I've thought of some ways to improve the scripts and the processes. Some of these suggestions are minor. Using multicast, however, would change the process significantly.

The main difficulty we had with the scripts I wrote was the status file that the various scripts use to keep track of the status of all the machines. Similar to the problems that large webservers encounter with their log files, this file turned out to be a bottleneck in the middle of JumpStarting a large number of machines. The multiple forks of the start script were in contention with each other in trying to update the status file. In the future, I will split the status file so that there is a separate file for each machine. This will eliminate the contention problems within start. It will also allow multiple operators to configure machines at the same time and not have to wait for access to the monolithic status file.

Another change that will improve the system is to split the start script into a true start script and a script that keeps track of the status of the JumpStart on each machine. These two scripts will run simultaneously during a rollout. On the other hand, helpdesk personnel or an operator could use just the basic start script to remotely JumpStart a single machine. This was done at the client after I left and made the rollouts proceed more smoothly.

The current method of tracking the progress of the JumpStart, using rsh to log into the machine and attempt to figure out what is going on, is somewhat error-prone. A nice addition would be to be able to view a real-time version of the log that is displayed on the client's screen while the JumpStart is progressing. This would allow the operator to see exactly what is going on and any errors that are displayed, without having to visit the machine.

Earlier I mentioned that we check for rogue bootparams servers but not rogue RARP or TFTP servers. Checking for rogue bootparams servers is simple because the rpcinfo program provides a feature that makes it easy to check for all bootparams servers on a subnet. Given an RPC service number and the -b flag, rpcinfo will broadcast a request on the subnet for that service and report all of the servers that respond. Any servers other than our designated boot server can be flagged as needing to be disabled. Unfortunately there isn't a similar program to check for rogue RARP or TFTP servers. Eventually I will write programs to do this.

In the long term I would like to investigate the possibility of using multicasting to distribute the disk images to the workstations. Each machine of a given kernel architecture gets an identical image, thus this is a prime candidate for multicasting. This would drastically reduce the NFS server bandwidth and capacity requirements as the NFS server would only need to serve out a couple of copies of the disk images (one for each kernel architecture) independent of the number of clients. This technology will be one to watch and keep in mind as it develops.

Making It Work For You

Hopefully by now I've interested some of you in improving your own JumpStart environments. I encourage you to take what I've done, modify it to fit the specifics of your site and then improve on things. I would welcome feedback on what worked, what didn't work and what you've done differently. Here are the major steps to implementing a system such as what I've described.

First, you'll want to build up your infrastructure. This includes placing boot servers on each subnet and ensuring sufficient NFS server capacity and bandwidth. If your clients are spread over multiple subnets, it would be preferable to have an NFS server on each

subnet. This reduces the amount of traffic that needs to cross a router. If your NFS servers have sufficient capacity, they can be attached to multiple subnets using several network interfaces.

The other infrastructure consideration is on the client end. Client workstations need to have standardized hardware. In addition, your environment should be designed in such a way that no data is stored on the clients. This is usually easy to accomplish and it makes the JumpStart process much simpler.

I would also highly recommend that you use some sort of distributed name service. The two most common are NIS and NIS+. The reason for this is that it ensures uniformity across all machines and provides a centralized location for making changes to the name service maps. This makes it easier and more reliable for the config script to check the accuracy of the name service maps and correct any errors. My scripts were written to support NIS but it should be fairly easy to convert them to an NIS+ environment.

The server that hosts the config and start scripts needs to have remote root login privileges on the clients, the boot servers and the NIS master (if applicable). This can be accomplished very insecurely using rsh and /.rhosts files or more securely using Secure Shell (SSH) and RSA keys. Login privileges are needed on the clients in order to gather information like kernel architecture, Ethernet address, etc. root privileges are needed to gather some of the required information. root login privileges are needed on the boot servers in order to make modifications to the links in /tftpboot and to start the RARP, bootparams and TFTP servers if necessary. root privileges are needed on the NIS master to edit the NIS maps and initiate map pushes.

Once all of these items are in place then you can work on integrating my scripts into your environment. As I mentioned previously, the scripts are written in Perl. Roughly 7000 lines of it, in fact. So you'll need to have Perl 5 available or lots of time to port the scripts to some other language. The scripts may be had from my web page at http://ofb.net/~jheiss/js/. I have released them under the GNU Public License.

The last recommendation I have is that you set up a test lab to test everything in. There are a lot of pieces to getting this all working and attempting a rollout without testing would be, well, crazy. I would recommend an isolated subnet with at least one machine of each model that exists in your environment. Thus if you have Sparc 5s, Sparc 20s and Ultra 1s as desktop machines you would have at least one of each in your test lab. This is especially important if you plan on using something like QuickJump as it is tricky to get the device trees right on each platform after restoring the dump files.

Author Information

Jason Heiss graduated from the California Institute of Technology in 1997 with a BS in Biology. He works for Collective Technologies as a systems management consultant, specializing in Solaris and Linux system administration, security and networking. He can be reached via email at <jheiss@colltech.com>. He can be reached via U.S. Mail at Collective Technologies; 9433 Bee Caves Road; Building III, Suite 100; Austin, TX 78733.

References

[Kas95] Kasper, Paul Anthony and Alan L. McClellan. *Automating Solaris Installations*. Prentice Hall, April 1995. ISBN: 013312505X.
This book provides detailed instructions on how to configure a Custom JumpStart. It is essential reading and reference for administrators new to configuring JumpStart. However, it does not discuss how to perform a rollout, especially on a large scale.

[Zub99] Zuberi, Asim, http://www.zdjournals.com/sun/s_sun/9902/sun9924.htm, "Jumpstart in a Nutshell," *Inside Solaris*, February 1999, pp 7-10.
A quick article with the steps to configure and use JumpStart with Solaris 2.6. As of Solaris 2.6, Sun rearranged the Solaris CD somewhat and this article shows the new paths you'll need to know. This makes a good companion to the book when working with newer versions of Solaris.

[SAIG] http://docs.sun.com/ab2/coll.214.4/SPARCINSTALL/, *Solaris Advanced Installation Guide*.
The official Sun documentation for JumpStart. Very good information and example scripts for configuring Custom JumpStart but again, it does not discuss how to perform a rollout.

[Won97] Wong, Brian L., *Configuration and Capacity Planning for Solaris Servers,* Prentice Hall, February 1997. ISBN: 0133499529.
This book provides excellent, detailed information on capacity planning for Solaris servers. The section relevant to sizing NFS servers for JumpStart begins on page 33.

[SI15834] Sun Infodoc 15834.
Instructions for disabling the installation of Maintenance Update patches.

Automated Client-side Integration of Distributed Application Servers

Conrad E. Kimball, Vincent D. Skahan, Jr., David J. Kasik – The Boeing Company
Roger. L. Droz – Analysts International

ABSTRACT

From the Single Glass Program Plan, dated August 18, 1997:

Vision: Provide BCAG users access to all applications and data needed to perform their respective jobs from a single desktop environment with acceptable levels of function, performance, and reliability.

The Single Glass program in the Boeing Commercial Airplane Group (BCAG) provides Engineering UNIX users the ability to access all required UNIX and PC applications and associated data via a standardized Common Desktop Environment (CDE) desktop.

The goal is to do this with enough performance, reliability, and transparency so that each engineer only needs one computer (i.e., a "Single piece of Glass") on their desk, thereby reducing the number of desktop devices required per engineer. There is an additional process benefit by increasing the amount of concurrent design and analysis possible.

The Single Glass desktop provides unified access to over 350 locally executed applications previously provided in a number of separate legacy environments from the shell and also within a common CDE look-and-feel developed through formal usability studies done in Seattle and Wichita. Currently running on over 5000 IBM RS6000 workstations worldwide for over 6000 users, Single Glass is designed to support IBM, HP, Sun, and SGI UNIX and NT workstations.

This paper describes some of the design decisions and project constraints that led Single Glass to decide to deliver a unified logical namespace to the clients that spans multiple physical servers. This was accomplished by automating creation of the required symbolic links to the many distributed file servers rather than simply building one monolithic "union of all the supplier trees".

This implementation has been proven to work consistently on AIX, HP-UX, Solaris, and Linux platforms.

Name Space

Reliable delivery of applications that have been developed in multiple organizations is a key Single Glass problem.

Boeing-written applications are organized via a standardized Boeing Common Directory Structure (CDS) analogous to many of the various file system naming standards in place elsewhere. A similar naming convention is in place for commercial-off-the-shelf (COTS) software.

The intent of CDS is to standardize the Boeing internal name space presented to the users, and to provide guidance to the software developers regarding where (and how) to install their software. Single Glass considers CDS to be the combination of public and private areas under one /boeing namespace [Figure 1].

In general, CDS requires the public namespace to consist solely of symbolic links to software components in the private namespace of that component's supplier. In addition, there are common directories to contain application configuration and log files.

The actual files that implement an application are encapsulated in a sub-tree of arbitrary complexity under /boeing/sw/<product>. Applications expose their public interface by installing symbolic links in known directories such as /boeing/bin, /boeing/man, and /boeing/lib.

Users include the stable set of public directories in $PATH, $MANPATH, and the like, and the symbolic links present there take care of providing access to the default (or specified) version of the application they wish to run from within the "private" portion of the /boeing tree.

A common script automates installing applications into the private namespace in the /boeing tree and modifying the appropriate public links so the "stable public path" delivers the current default version of the applications to the user.

Typical Software Installation

Figure 2 shows a minimal installation of version a02 of a hypothetical product "foo" consisting of one executable with a corresponding manual page.

The common installer tool installs the actual software in the private directory of the application supplier, and creates the appropriate links in the public directories which are referenced in every user's $PATH and $MANPATH.

Both version dependent and independent links are present in the public directories under /boeing to permit the users to run either the default version or a user-specified version of each product. The products are compiled and configured to use the version-dependent paths so that multiple versions of any product may be present at the same time. Users reference the "stable" path /boeing/bin/foo for the current default version of the product, or can choose to reference an absolute version-specific path of /boeing/bin/foo_a02 to get to version a02 of the product (for example).

Similarly, they can view the manual page for the product by specifying the default product name "man foo" or the version-specific name "man foo_a02".

The installer always uses fully qualified (rather than relative) pathnames when creating symbolic links in order to permit the physical implementation of a /boeing tree to span multiple filesystems (glued with automounter or the like) if necessary.

Multiple Fileservers

Rather than split a single /boeing tree onto multiple fileservers, Single Glass does essentially the opposite. We automate assembly of a single logical tree on the client side from multiple whole application fileserver trees provided by a number of organizations within Boeing [Figure 3].

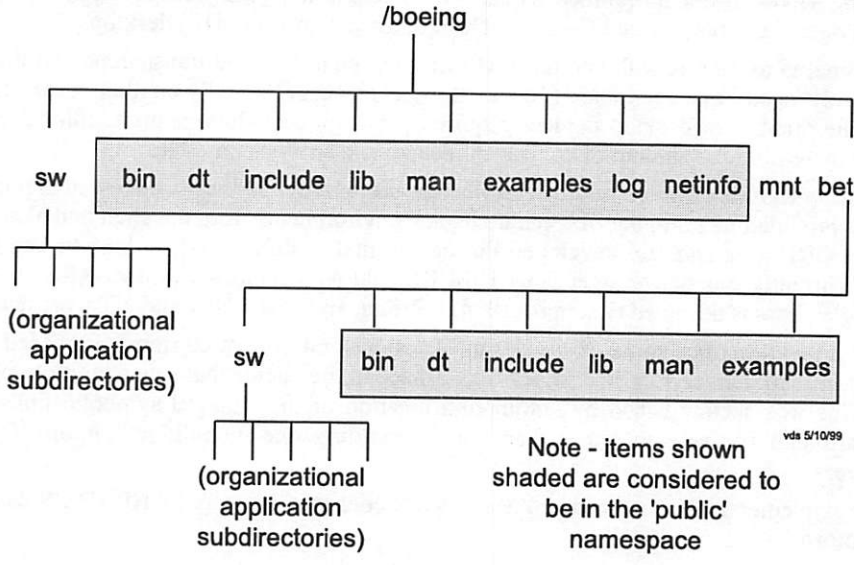


Figure 1: Boeing common directory structure (excerpt).

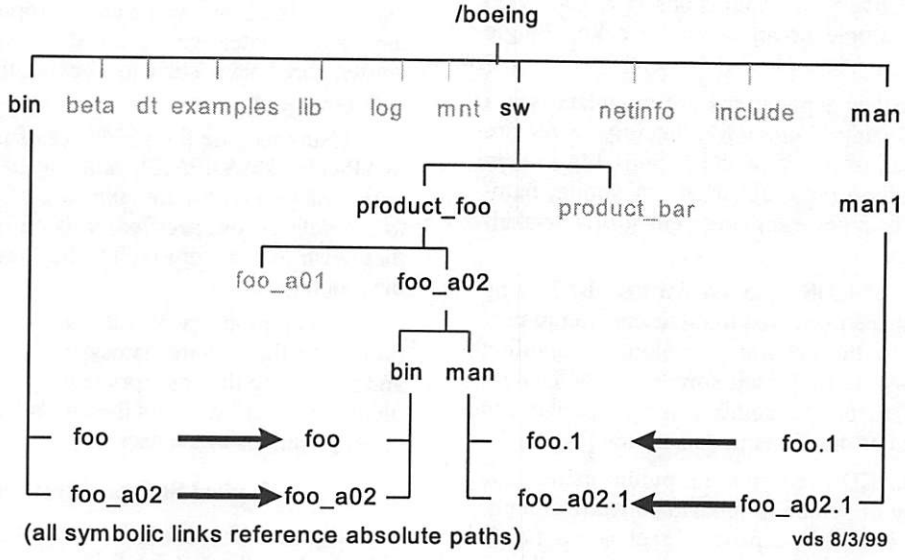


Figure 2: Typical product installation in common directory structure.

Single Glass currently supports applications coming from at least seven different application providers, many of whom supply their own fileserver capacity to be mounted by the clients. There is no (current) effective way to pool these servers (and budgets), nor is this considered wise strategically as scale increases.

Traditionally, the Single Glass project would have assembled a monolithic union of the various application provider unique /boeing trees and presented that unified tree to the clients through one project-wide /boeing mount of that assembled tree. This was impractical for a number of reasons.

- There are limits related to "how big a fileserver is possible or wise".
- Our implementation is being done in a staged manner over a long period of time rather than as a "big bang" event, and procurement of interim fileservers to provide sufficient capacity during the transition period of over 24 months was impractical.
- The very act of performing an integration into one physical /boeing tree structure tends to induce territorial and process conflict among the contributing application communities with respect to managing application content and versioning.
- We assume a heterogeneous audience that uses multiple versions of the same application. Our distributed user community has diverse views

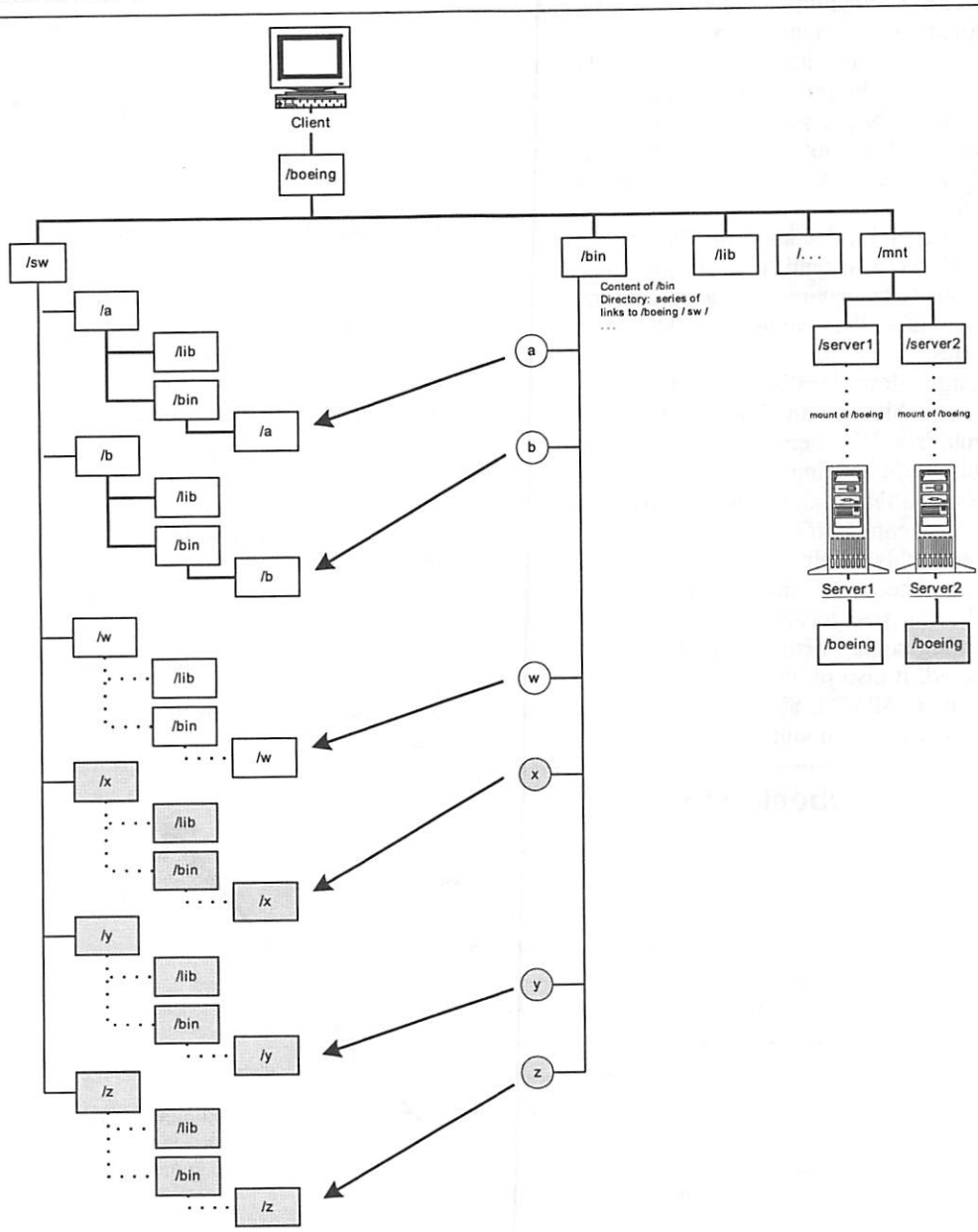


Figure 3: Client-side view of application servers.

regarding the need to standardize on one version of each product or not, and agreeing on a single version of any application is difficult to implement.

- Our application suppliers are used to a great deal of autonomy in how they develop, test, and deliver their software. Each supplier tends to have their own unique installer tool that installs the software into /boeing in a CDS compliant manner. This autonomy led us to a solution where we would need to assemble an integrated union of multiple independently "valid" supplier-provided /boeing trees.
- The requirement to support the legacy Boeing Common Directory Structure (which mandates extensive use of symbolic links) made a simple automounter solution impossible.
  Automounter would have been a fine solution for assembling the private portions of the /boeing tree (each of the subdirectories under /boeing/sw) but it cannot assemble the extensive sets of symbolic links in the public portions of the tree.
  By creating a local composite /boeing/bin on the client, we also tend to avoid multiple network transits by having a local /boeing/bin in $PATH rather than multiple NFS mounted directories.
- Our design allows locally installed /boeing tree software to blend with /boeing tree software from multiple NFS servers. This required the assembly of the /boeing tree to be moved to the client-side so that each client can have its own unique local content if required.

Client-side assembly minimizes disk space and software distribution headaches involved with maintaining the entire /boeing tree locally on the client system, while permitting some clients to have local installations if required. It also provides some caching effect as all the items in $PATH, $MANPATH, and the like are resident locally as symbolic links.

Accordingly, the decision was made to perform client-side assembly of whole /boeing trees with automated tools.

Automating the Client-side Assembly Process

Single Glass builds a composite client-site /boeing tree consisting of (generally minimal) client-side local content and many symbolic links pointing to the various remote servers mounted under /boeing/mnt.

This is done on a first come, first served basis (alphabetically), under a local /boeing/mnt, with local content having the highest precedence [Figure 4].

The easiest way to understand this process is to imagine laying several CDS trees on top of each other. Ideally, the contributing trees will merge with no conflict. In practice, when conflicts do occur, they are resolved in favor of the first contributor. The local workstation takes highest priority, followed by the servers as they appear in alphabetical order in the /boeing/mnt directory.

The various /boeing trees are dynamically "layered" on a first-come-first-served basis by a Single Glass-provided perl program that "integrates" the multiple server trees into a consistent client-side view.

This process takes between 90-180 seconds and creates approximately 5200 required symbolic links on the client-side. Virtually all the actual time is spent calculating what directories and symbolic links to add, delete, or modify locally on the client.

Since server-side changes are (hopefully) relatively infrequent, the integration script is only run once nightly via cron on all workstations or at the end of the system boot sequence. In this way, a simple reboot will restore the client to "last known-good" configuration.

The integration script adds, deletes, and modifies the appropriate links on the client so the workstation can reference the combination of locally installed software and software residing on several code servers through a single unified /boeing.

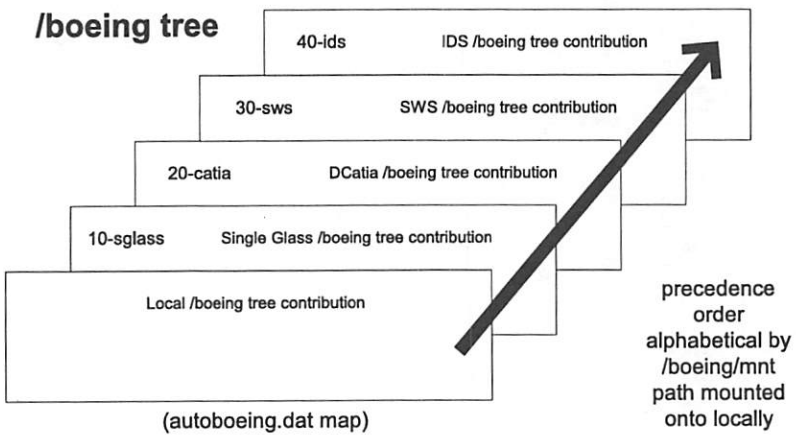


Figure 4: Layered application servers.

The integration script:

- Reads the master automounter map and identifies all trees that could be (auto)mounted under /boeing/mnt.
- Reads a pre-computed inventory file (a compressed "ls -AlRc") from each of those servers for efficiency reasons.
  If no inventory file is available, it walks the tree to calculate an inventory, with considerable network and server load due to the size of the /boeing trees.
- Compares the local /boeing tree with the union of the remote tree inventories.
- Adds, deletes, modifies the appropriate links and directories to create the combined union of all the trees on the local client /boeing filesystem.
- Cleans up any obsolete links, removes any local /boeing tree subdirectories that are empty.

This assembly process takes place at the individual workstation level, so that all users of a particular workstation have an identical view of the available applications.

Similarly, since workstations use the same automounter maps, each workstation in a particular NIS domain is integrated identically, yielding identical client-side configurations on each workstation.

Users can of course choose to alter their personal defaults by customizing (at their own risk) their personal dot files. In case the user's customizations make their account unusable or unstable, we provide a CDE action to quickly move their modifications aside and reset the account to a known good initial configuration.

How the Script Works Internally

CDS is organized around "products" and "portfolios", which are essentially collections of products developed by one organization.

The integration script first identifies and builds symbolic links to all "products" and "portfolios" located under /boeing/sw [Figure 5 – item 1]. The client can now see the "private" or "physical" installation tree of all software packages in the network.

Next, the script merges the "public interface directories." The corresponding directories are searched on each contributing tree. Links that point into the private directory structure of a "product" contributed by a given server are reproduced in the client's public interface directory [Figure 5 – item 2].

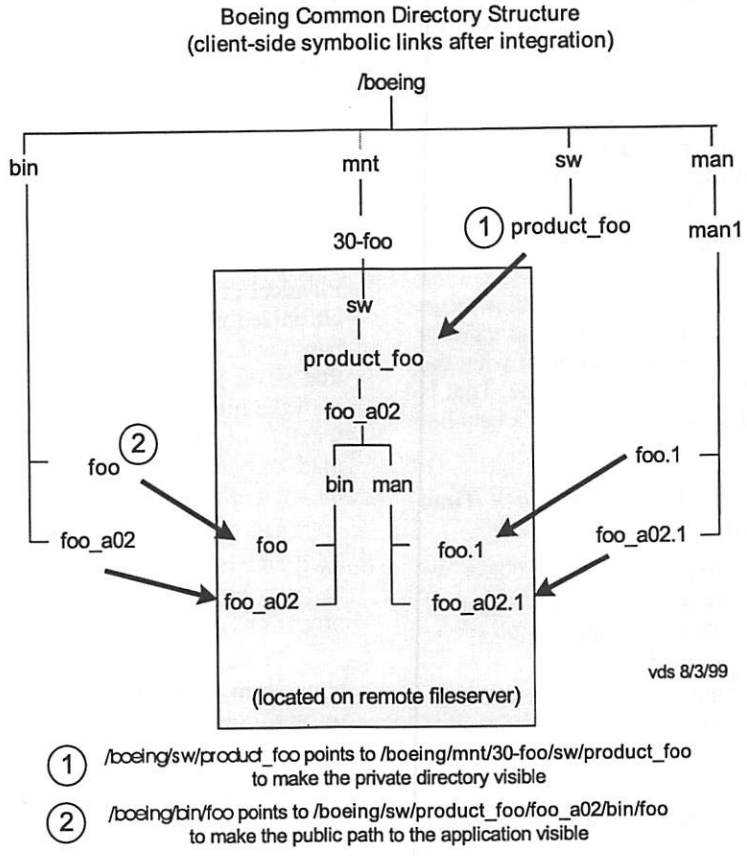


Figure 5: Client-side view after integration.

The script recursively descends the public interface, reproducing links to files, until a link to a directory in the private namespace is encountered.

Last, the script, if requested, will delete local links that did not come from one of the contributing /boeing tree servers unless they meet one of the following conditions:

- are located in /boeing/netinfo on the local workstation
- are located in /boeing locally, and also listed in the /boeing/netinfo/preserve.aix file (so called "precious" files as mentioned in the actual perl code)

When the client is unable to integrate a particular tree due to a remote server-down situation, the pre-existing links from that server are assumed to still be appropriate and are left intact. Links pointing to a server that returns a permission-denied message on the automounter attempt to mount are considered no longer valid and are deleted.

In addition, the integration script will log, but not create, any links that are calculated to be "dangling", so that the client-side configuration only reflects applications that will (presumably) be functional after the integration is complete.

STDOUT and STDERR are captured to log files in /tmp on the local workstation that can be used for debugging purposes.

It is important to note that the integration script is indeed an integrator, not an installer. It blindly believes that the application provider has ensured that their software installs so that the public/private directory separation is in place, and equally blindly copies the link values present in those remote trees to the client side.

While the integration script has some knowledge of the requirements of the CDS standard and can log discrepancies it finds, it has no knowledge of what the link values in the public directories should be. That is the responsibility of the application provider's installer tool.

Caching Server Inventory To Reduce Clock Time And Network Load

Early in the integration phase of Single Glass, we identified that there can be a tremendous amount of traffic generated to inventory a large application server once, let alone 5000 times in parallel.

Pre-computing a compressed inventory of the fileservers, and having the clients "quickly read a remote file then work totally locally" mitigates this cost.

The inventory script only writes a new inventory file into place if there have been changes made that the integrate script "cares about". This permits the served inventory files to be used as timestamp sentinels for comparison to permit "lazy (i.e., do it only if needed) integration" client-side.

This pre-computed inventory eliminated over 99% of the network load and 50% of the clock time needed to synchronize the Single Glass clients with the /boeing tree servers (vs. our initial implementation).

Total NFS load for a full integration is currently approximately 1300 NFS operations as follows:

Nfsstat(1M) call	Number of Calls
Getattr	27
Lookup	501
Access	75
Readlink	2
Read	671
Readdir+	11
Fsinfo	5

Integration takes 90-180 seconds depending on client CPU speed (IBM model 42T and 43P workstations).

Randomized "Lazy" Integration

In all cases, the integration script only integrates the clients when there are changes to be made (so-called "lazy" integration).

This is established by examining the timestamps on the local log files versus the inventory files on each of the servers.

If any of the remote server inventories is newer than the local log file, the client does a full integration to synchronize itself to the union of the remote server configurations.

Since the integration script is called from an identical crontab entry on 5000 clients that have synchronized clocks, a client-side ksh script and accompanying C program serve as a randomizer to ensure that all the systems don't "wake up at once" and overload the file servers or networks. Both files are stored locally in on the workstation to minimize network load and are the only local content required to make the integration occur.

The compiled program determines, based on the TCP address of the local system, how many seconds to sleep before calling the integration script with the proper options.

The current settings are intended to have all systems sleep for no more than one hour before waking up and doing the actual /boeing tree integration, but this value is also configurable at run-time if needed.

Lastly, client-side integration can be "forced" to supersede a calculated answer of "no integration required".

Ensuring The Clients Are Up To Date After Reboot

Frequently workstations are relocated, or are booted from alternate drives for special beta testing

and the like, and thus miss their once-per day scheduled cron-based integration. To ensure that the systems are always in the most up-to-date possible state after a reboot, the integrate script is called out of the client boot sequence after automounter is up.

Non-technical Challenges Of Client-side Integration

There are of course downsides to client-side integration that tend to be more cultural (Fear-Uncertainty-Doubt) than technical in nature.

- There is the need to synchronize multiple server installations across many sites (but of course doing so on 25 fileservers is better than doing so on 5000 clients).
- The traditional approach to providing content consisting of a large number of applications is to provide an equally large file server rather than our distributed logical approach.
  Management, especially when migrating from a mainframe model, often questions the reliability and availability of such a distributed approach.
  Fortunately, recent problem report call analysis shows average measurable downtime of less than one unscheduled hour per month per workstation.
- Our distributed customer base has widely varying definitions of "acceptable risk", "required testing", and ability to be the recipient of change in their computing environments.
  Our distributed logical file server model permits application versions to be updated asynchronously to each other at each application user community's own pace. This causes certain "culture collisions" among the different user communities regarding how many combinations of versions need to be tested against each other.
- Client-side integration introduces one more (final) step in the client workstation build/bringup procedure. This has required some changes to the workstation system assembly procedures and training. Conclusions

The architecture and tools described above have permitted us to logically layer a number of existing large application fileserver implementations into a consistent client-side view of the union of the possible applications available for use on over 5000 workstations worldwide.

This has been done within a well-defined common directory structure and associated delivery system implementation constraints and rules. Our approach has succeeded in not affecting the implementation of the legacy systems or requiring change in the existing software installation tools used by the internal software developers.

Rather than increasing total project cost, this implementation has lowered the cost of entry into Single Glass for a new application provider, as we can avoid the costs (and potential battles) of requiring potential application providers to be "assimilated" into the project.

The result of our implementation is an architecture that has proven to be portable, extensible, reliable, and supportable worldwide at a 5000 workstation scale.

Futures

Given our heavy reliance on NFS, we need to investigate NFS caching to improve performance and minimize traffic.

Our cached inventory files on the fileservers contain quite a bit more information than the integrate script actually needs. We believe shrinking the precomputed inventories to the minimum needed for the integrate script to do the job would result in load and time improvements.

While the various links created by the integration script are resident client-side, CDE reads through the symbolic links at user login time when building the application manager desktop, causing a several second delay in logging in as each fileserver providing pieces of the assembled /boeing/dt is automounted. Bringing the CDE tree assembly process fully client-side by copying the CDE related files (rather than just links) into the more traditional /etc/dt directory during the integration process on the workstation should speed up login time noticeably.

The users always expect their systems to be always "up" and always "up-to-date". In a worldwide 24x7x365 company, this is a major long-term issue to overcome.

Any fileserver-side changes do not take effect until the server is re-inventoried, and the client is re-integrated (which normally happens just on reboot, or nightly via cron). Re-integrating the client while a user is logged in can have adverse affects to the user's session (i.e., changing the default version of "foo" while they have a session of the old version running).

We need to be able to have the clients check "should I catch up" more often (or be notified automatically) and have them integrate "on the fly" without affecting any logged in users or running jobs in any way.

Author Information

Conrad Kimball <Conrad.Kimball@boeing.com> is an Associate Technical Fellow of the Boeing Company. He is a co-architect for the Puget Sound Single Glass program, and is a member of the Boeing-wide computing delivery system Technical Leadership Team and Technical Planning Board. Conrad holds a Master of Software Engineering degree from Seattle University.

Vince Skahan <Vince.Skahan@boeing.com> is a System Design & Integration Specialist at the Boeing

Company. He holds a B.S. in Chemical Engineering from Drexel University, and has been doing large scale heterogeneous unix administration since 1987.

David J. Kasik <David.J.Kasik@boeing.com> is a Technical Fellow of the Boeing Company. He is a co-architect for the Puget Sound Single Glass program and acts as the Geometry and Visualization architect for Boeing Commercial Airplanes.

Roger Droz' <Roger.Droz@seaslug.org> career has spanned a broad range of computing environments as a designer and programmer – from 8 bit embedded systems to the enterprise scale of Single Glass; from operating system kernel and device drivers to scientific and commercial applications. Roger holds a Masters degree in Electrical Engineering from Washington State University.

References

Linux filesystem standard online at http://www.pathname.com/fhs/.

Common Directory Structure and Supporting Environments for BCAG Unix Systems, Boeing internal document D6-81580 dated 3/10/95.

Kasik, D., Kimball, C., Felt, J., Frazier, K. *A Flexible Approach to Alliances of Complex Applications*, presented at ICSE'99, http://sunset.usc.edu/icse99.html.

Bell, John D., *A Simple Caching File System for Application Serving*, http://www.usenix.org/publications/library/proceedings/lisa96/full_papers/jbell4.html.

Hauser, C., *Speeding Up UNIX Login by Caching the Initial Environment*, http://www.usenix.org/publications/library/proceedings/lisa94/hauser.html.

Ph. Defert, E. Fernandez, M. Goossens, O. Le Moigne, A. Peyrat, I. Reguero, *Managing and Distributing Application Software*, http://www.usenix.org/publications/library/proceedings/lisa96/full_papers/reguero/reguero.txt.

Furlani, John L., Osel, Peter W., *Abstract Yourself With Modules*, http://www.usenix.org/publications/library/proceedings/lisa96/pwo.html.

Wong, Walter C., *Local Disk Depot – Customizing the Software Environment*, http://www.usenix.org/publications/library/proceedings/lisa93/wong.html.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *;login:*, the Association's magazine, published eight–ten times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and C++, book and software reviews, summaries of sessions at USENIX conferences, and Snitch Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, via the USENIX Online Library on the World Wide Web.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as object-oriented technologies, security, operating systems, electronic commerce, and NT – as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Discount on BSDI, Inc. products.
- Discount on all publications and software from Prime Time Freeware.
- Savings (10-20%) on selected titles from Academic Press, Morgan Kaufmann, New Riders/Cisco Press/MTP, O'Reilly & Associates, OnWord Press, The Open Group, Sage Science Press, and Wiley Computer Publishing.
- Special subscription rates for Cutter Consortium newsletters, *The Linux Journal, The Perl Journal, IEEE Concurrency, Server/Workstation Expert*, *Sys Admin Magazine*, and all Sage Science Press journals.

Supporting Members of the USENIX Association:

C/C++ Users Journal
Cirrus Technologies
Cisco Systems, Inc.
CyberSource Corporation
Deer Run Associates
Greenberg News Networks/MedCast Networks
Hewlett-Packard India Software Operations
Internet Security Systems, Inc.
JSB Software Technologies
Lucent Technologies
Macmillan Computer Publishing, USA
Microsoft Research
MKS, Inc.
Motorola Australia Software Centre
NeoSoft, Inc.
New Riders Press
Nimrod AS
O'Reilly & Associates Inc.
Performance Computing
Questra Consulting
Sendmail, Inc.
Server/Workstation Expert
TeamQuest Corporation
UUNET Technologies, Inc.
Web Publishing, Inc.
Windows NT Systems Magazine
WITSEC, Inc.

Sage Supporting Members:

Atlantic Systems Group
Collective Technologies
D. E. Shaw & Co.
Deer Run Associates
Electric Lightwave, Inc.
ESM Services, Inc.
GNAC, Inc.
Macmillan Computer Publishing, USA
Mentor Graphics Corp.
Microsoft Research
MindSource Software Engineers
Motorola Australia Software Centre
New Riders Press
O'Reilly & Associates Inc.
Remedy Corporation
RIPE NCC
SysAdmin Magazine
Taos Mountain
TransQuest Technologies, Inc.
Unix Guru Universe

For further information about membership, conferences or publications, contact:
USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA.
Phone: 510-528-8649. Fax: 510-548-5738.
Email: *office@usenix.org.*
URL: *http://www.usenix.org.*

ISBN 1-880446-25-1